N° d'ordre : 1018

# THÈSE

**présentée à**

L'UFR des Sciences et Techniques
Université de Franche-Comté

**pour obtenir le**

**GRADE DE DOCTEUR DE L'UNIVERSITÉ DE FRANCHE-COMTÉ**
**Spécialité Automatique et Informatique**

# Contribution au processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers des langages à objets

par
Jean-Christophe VOISINET

Soutenue le 14/09/2004 devant la Commission d'Examen :

| | |
|---|---|
| Rapporteurs | M. Didier Bert, Chargé de Recherche du CNRS au LSR-IMAG - Grenoble |
| | Mme Jeanine Souquières, Professeur à l'Université de Nancy II |
| Examinateurs | M. Bruno Legeard, Professeur à l'Université de Franche-Comté - Besançon |
| | M. Jean-Louis Lanet, Ingénieur de recherche associé - INRIA - Sophia-Antipolis |
| Directeur de thèse | M. Jacques Julliand, Professeur à l'Université de Franche-Comté - Besançon |
| | M. Bruno Tatibouët, Maître de conférences à l'Université de Franche-Comté - Besançon |

# Remerciements

Je tiens à remercier tout d'abord mon jury de thèse. Merci à :
- M Bruno Legeard pour avoir accepté de présider mon jury de thèse,
- M$^{me}$ Jeanine Souquières et M Didier Bert pour m'avoir fait l'honneur de rapporter cette thèse. Leurs remarques m'ont permis d'améliorer la qualité de ce document,
- M Jean-Louis Lanet pour ses commentaires sur les développements pratiques relatifs au domaine industriel de ces travaux.
- M Bruno Tatibouët et M Jacques Julliand qui ont dirigé cette thèse, pour leur disponibilité, leur soutien, leur confiance et leur patience.

Merci à tous les membre du LIFC en particulier les colocataires du Sauna 432 C, Tus, Fred, Niklas, Bruno, Sylvain et Suphakit, avec qui travailler fut toujours un plaisir en raison de leur convivialité et disponibilité.

Je tiens aussi à remercier ma famille qui m'a soutenu et encouragé dans les moments agréables mais aussi désagréables ! Je pense plus particulièrement à ma mère, mes grands parents, oncles et tantes, Rod, Kiki, Rourou, Zab, Flo, Nath, Nico et GG.

Un gros merci à mes amis de Besançon pour leur bonne humeur de tous les jours : Isa, RV, Poney, Weï, Seb, Yan, GroGG, Jean-Claude, Petite Blonde, Djeff, Tof, Cé, Marseillais, Hélène, Petit Jérôme etc...

Egalement merci à mes amis d'enfance d'Evette-Salbert city plage, Flash, Val, Céline, Christophe, Mars, Lydie, Flox, Myrette, Nath, Romain, Lepage...

Enfin j'ai une petite pensée pour les piliers de comptoir de l'Aqua et les adorateurs des dieux Taros et Belotus...

# Table des matières

**I Présentation** **23**

**1 Introduction** **25**
1.1 Contexte . . . . . 26
1.2 Choix de la méthode formelle . . . . . 27
1.3 Contributions et objectifs de recherche . . . . . 27
1.4 Structure du document et guide de lecture . . . . . 28
1.4.1 Guide de lecture . . . . . 29
1.4.2 Publications issues de ce travail de thèse . . . . . 29

**2 La méthode B** **31**
2.1 Introduction . . . . . 31
2.2 Les méthodes formelles . . . . . 31
2.3 La méthode B . . . . . 32
2.3.1 Fondements de la méthode B . . . . . 32
2.3.2 Raffinement et implémentation . . . . . 33
2.3.3 Preuve . . . . . 33
2.3.4 Structure générale d'une machine abstraite . . . . . 34
2.3.5 Substitutions généralisées . . . . . 35
2.3.6 Théorie des ensembles et typage . . . . . 36
2.3.7 Obligations de preuve . . . . . 38
2.3.8 Les mécanismes de composition . . . . . 40
2.4 Conclusion . . . . . 43

**II Extraction de vues UML** **45**

**3 Contexte et motivations** **47**
3.1 Utilisation conjointe des méthodes formelles et semi-form-elles . . . . . 47
3.2 Motivations . . . . . 48
3.3 Contenu de la deuxième partie . . . . . 49

**4 La notation UML** **51**
4.1 Introduction . . . . . 51
4.2 Généralités . . . . . 51
4.3 Diagramme de classes . . . . . 52
4.3.1 Classes, attributs et méthodes . . . . . 52
4.3.2 Généralisation . . . . . 53

4.3.3 Association, qualification, multiplicité, agrégation et composition . . . 53
4.3.4 Qualification des associations . . . 54
4.4 Diagramme d'états-transitions . . . 55
4.4.1 Etats . . . 55
4.4.2 Transitions . . . 56
4.4.3 Concurrence . . . 56
4.5 Diagramme de séquence . . . 57
4.5.1 Les objets . . . 57
4.5.2 Messages . . . 58
4.5.3 Temps d'exécution . . . 58
4.5.4 Conditions . . . 58
4.5.5 Pseudo-code . . . 58
4.5.6 Contraintes temporelles . . . 59
4.6 Diagramme d'activités . . . 59
4.6.1 Les objets . . . 60
4.6.2 Exécutions concurrentes . . . 60
4.6.3 Points de choix et conditions . . . 60
4.7 Le langage OCL . . . 61
4.7.1 Le contexte . . . 61
4.7.2 Les types de base et les opérateurs . . . 62
4.7.3 Les types et sous-types de classificateurs . . . 62
4.7.4 Les propriétés des objets . . . 62
4.7.5 Les valeurs antérieures . . . 62
4.7.6 La navigation vers les extrémités d'association . . . 62
4.7.7 Les collections . . . 63
4.7.8 Les classes-associations . . . 64
4.7.9 Les variables . . . 64
4.8 Conclusion . . . 64

**5 Etat de l'art : Dérivation d'UML vers B 65**
5.1 Introduction . . . 65
5.2 Dérivation à partir de diagrammes de classes UML . . . 65
5.2.1 Dérivation des classes . . . 66
5.2.2 Attribut . . . 67
5.2.3 Opération . . . 67
5.2.4 Association . . . 68
5.2.5 Généralisation . . . 69
5.3 Extension de la dérivation des diagrammes de classes . . . 70
5.3.1 Motivations . . . 71
5.3.2 Dérivation des prototypes et appels d'opérations . . . 71
5.3.3 Classification et organisation des opérations . . . 72
5.4 Dérivation à partir de diagrammes d'états-transitions . . . 72
5.4.1 Dérivation de statecharts en machines abstraites . . . 72
5.4.2 Extension de la dérivation associée au diagramme de classes . . . 76
5.5 Traduction des expressions OCL en B . . . 81
5.5.1 Types de base prédéfinis OCL . . . 82
5.5.2 Type collection . . . 82

5.5.3 Expression let ... in ... 82
5.5.4 Expression de post-condition 83
5.6 Extraction de systèmes de transitions étiquetées 83
5.6.1 Système de transitions étiquetées 83
5.6.2 Principe de construction d'un système de transitions fini 84
5.6.3 Enumération des états 84
5.7 Conclusion 85

**6 Extraction de diagrammes de classes 87**
6.1 Introduction 87
6.2 De la spécification B au diagramme de classes 87
6.2.1 Les machines abstraites 87
6.2.2 Les variables entières et booléennes 87
6.2.3 Les ensembles simples 89
6.2.4 Les ensembles de couples 91
6.2.5 Utilisation de stéréotypes 95
6.3 Un exemple simple 96
6.3.1 Etude de cas : Inscription d'étudiants à un diplôme 96
6.3.2 Modélisation UML correspondante 97
6.4 Mécanismes de composition 99
6.4.1 Visibilité à travers les clauses SEES et IMPORTS 99
6.4.2 Clause EXTENDS, INCLUDES et PROMOTES 99
6.5 Comparaison avec d'autres propositions de traduction 100
6.5.1 Introduction 100
6.5.2 Etude de cas : système de contrôle d'accès aux batiments 100
6.5.3 Les traductions en UML 101
6.5.4 Comparaisons de ces deux approches 101

**7 Génération de contraintes OCL 103**
7.1 Introduction 103
7.2 Réécriture en OCL des invariants B 103
7.2.1 Introduction 103
7.2.2 Tableaux d'équivalence B vers OCL 104
7.2.3 L'exemple du Scheduler 106
7.2.4 Les problèmes rencontrés 108
7.3 Réécriture en OCL des opérations B 109
7.3.1 Présentation des opérations B 109
7.3.2 Transformation en OCL d'une opération B 109

**8 Extraction de diagrammes d'états-transitions 113**
8.1 Introduction 113
8.2 Limites de l'extraction 113
8.3 Grammaire du noyau du langage utilisé 113
8.4 Principes généraux de traduction 115
8.5 La structure 115
8.6 Les substitutions simples 116
8.7 Initialisation, opérations et événements 117

8.8 Les substitutions de composition . . . 118
8.8.1 Substitution *Bloc* . . . 118
8.8.2 Substitution *Sélection* . . . 119
8.8.3 Substitution *Précondition* . . . 119
8.8.4 Substitution *Choix borné* . . . 119
8.8.5 Substitution *Condition* . . . 120
8.8.6 Substitution *Condition par cas* . . . 121
8.8.7 Substitution *Choix non borné* . . . 121
8.8.8 Simplifications par évaluation des expressions . . . 122
8.8.9 Variables concurrentes . . . 123
8.8.10 Paramètres . . . 123
8.9 Calcul de diagrammes extraits de substitutions imbriquées . . . 124
8.10 Calcul du diagramme significatif par sélection de variables . . . 125
8.11 L'exemple du Robot . . . 127
8.11.1 Présentation . . . 127
8.11.2 Principes de fonctionnement . . . 127
8.11.3 Diagramme d'états transitions des machines abstraites . . . 128
8.12 Comparaison avec d'autres méthodes . . . 134

**9 Diagrammes de séquence et d'activités 139**
9.1 Introduction . . . 139
9.2 Diagramme de séquence . . . 139
9.2.1 Les objets . . . 139
9.2.2 Les messages . . . 140
9.2.3 Les paramètres et valeurs de retour des opérations . . . 141
9.2.4 Les conditions . . . 142
9.2.5 Les boucles dans les diagrammes de séquence . . . 142
9.3 Diagrammes d'activités . . . 143
9.3.1 Les acteurs du diagramme d'activités . . . 143
9.3.2 Initialisation, opération et états d'acceptation . . . 144
9.3.3 Séquentialité . . . 145
9.3.4 Expressions conditionnelles . . . 146
9.3.5 Les boucles . . . 147
9.3.6 Appel d'opérations, paramètres et valeurs de retour . . . 148

**10 Conclusion sur l'extraction de vues graphiques 151**
10.1 Extraction des informations statiques . . . 151
10.1.1 Extraction des diagrammes de classes . . . 151
10.1.2 Génération des contraintes OCL depuis les propriétés complexes de l'invariant . . . 152
10.1.3 Synthèse de l'extraction des données statiques . . . 152
10.2 Extraction des informations dynamiques . . . 152
10.2.1 Extraction des diagrammes d'états-transitions . . . 152
10.2.2 Extraction des diagrammes de séquence et d'activités . . . 152
10.2.3 Synthèse de l'extraction des informations dynamiques . . . 153

**III Génération de code source pour des langages à objets 155**

**11 Génération de code à partir de spécifications formelles 157**
11.1 Introduction . . . 157
11.2 Projet BOM . . . 157
11.3 Objectifs du projet BOM . . . 158
11.4 Objectif et contribution de mes travaux de thèse . . . 158
11.5 Etat de l'art . . . 160
11.5.1 Projet BOM : de B vers C . . . 160
11.5.2 De B vers OCaml . . . 160
11.5.3 De ObjectZ vers C++/Java . . . 162
11.5.4 De VDM++ vers C++ et Java . . . 163
11.6 Conclusion . . . 163

**12 Génération de code Java à partir de B 165**
12.1 Introduction . . . 165
12.2 Principes généraux adoptés . . . 165
12.2.1 Le point de départ de la traduction : le B0 . . . 165
12.2.2 Exemple élémentaire de machine abstraite et d'implémentation . . . 166
12.2.3 Schéma de conversion des machines . . . 166
12.3 La représentation d'un projet B . . . 167
12.4 La représentation des machines en Java . . . 167
12.4.1 Machines B et schémas de traduction Java . . . 167
12.4.2 Exemples . . . 168
12.5 Les types . . . 168
12.5.1 Les types de base du langage B . . . 168
12.5.2 Equivalence entre types B et types Java . . . 174
12.6 La Traduction des données . . . 174
12.6.1 Remarques . . . 174
12.6.2 Les ensembles . . . 175
12.6.3 Les constantes concrètes . . . 176
12.6.4 Les variables concrètes . . . 177
12.6.5 Les intervalles . . . 177
12.6.6 Les tableaux . . . 178
12.6.7 Les paramètres des machines . . . 181
12.7 Les opérations . . . 182
12.7.1 La traduction des opérations B et des paramètres d'entrée . . . 182
12.7.2 Le problème des résultats multiples . . . 184
12.7.3 Un exemple . . . 185
12.7.4 Retour sur la sémantique du passage de paramètres en B . . . 185
12.8 Les substitutions . . . 188
12.8.1 Introduction . . . 188
12.8.2 La substitution *Devient égal* . . . 189
12.8.3 Substitution appel d'opération . . . 189
12.8.4 Substitution *Séquencement* . . . 191
12.8.5 Substitution *Bloc* . . . 191
12.8.6 Substitution *Identité* . . . 191

12.8.7 Substitution *Variable locale* . . . 192
12.8.8 Substitution *Condition* . . . 192
12.8.9 Substitution *Condition par cas* . . . 195
12.8.10 Substitution *Boucle tant que* . . . 195
12.8.11 Substitution *Assertion* . . . 196

**13 Vers une traduction pour JavaCard** **199**
13.1 Introduction . . . 199
13.2 La plate-forme JavaCard . . . 199
13.2.1 Les spécificités de la plate-forme JavaCard . . . 199
13.2.2 Structure d'une applet pour JavaCard . . . 201
13.3 L'exemple du porte-monnaie électronique . . . 203
13.3.1 Présentation . . . 203
13.3.2 Les machines B . . . 203
13.3.3 La traduction en Java . . . 204
13.4 Génération de code JavaCard . . . 205
13.4.1 L'applet JavaCard du porte-monnaie électronique . . . 205
13.4.2 Le processus de développement JavaCard semi-automatique . . . 205

**14 Conclusion sur la génération de code objet à partir de spécifications B** **209**
14.1 Traduction de spécifications B en langage à objets . . . 209
14.2 Evaluation . . . 210

**IV Outils et applications** **211**

**15 La plate-forme jBTools** **213**
15.1 Introduction . . . 213
15.2 La plate-forme commune . . . 214
15.2.1 Architecture et principe de fonctionnement . . . 214
15.2.2 Le TypeChecker . . . 214
15.2.3 Le Contexte B . . . 215
15.2.4 Le format XML . . . 215
15.2.5 La traduction du typage en XML . . . 215
15.2.6 La B Object Library (BOL) . . . 215
15.3 Les outils . . . 217
15.3.1 L'outil B2UML . . . 218
15.3.2 Traducteur vers Java/C# . . . 221
15.4 Conclusion . . . 221

**16 Application et mise en oeuvre** **223**
16.1 Introduction . . . 223
16.2 Cahier des charges . . . 223
16.2.1 Présentation . . . 223
16.2.2 Propriétés du système . . . 224
16.3 Spécifications du projet B . . . 224
16.3.1 Le composant *Feu* . . . 224

16.3.2 Le composant *CARREFOUR* . . . . . 225
16.3.3 Composant principal . . . . . 226
16.4 Extraction de vues UML . . . . . 227
16.4.1 Diagramme de classes . . . . . 227
16.4.2 Contraintes OCL . . . . . 228
16.4.3 Diagramme d'états-transitions . . . . . 229
16.4.4 Diagramme de séquence . . . . . 231
16.4.5 Diagramme d'activités . . . . . 231
16.5 Traduction en Java . . . . . 232

**V Conclusion** **237**

**17 Conclusion et perspectives** **239**
17.1 Contributions . . . . . 240
17.1.1 Extraction de vues UML . . . . . 240
17.1.2 Traduction vers les langages à objets Java et C# . . . . . 241
17.1.3 La plate-forme jBTools . . . . . 241
17.1.4 Synthèse des contributions . . . . . 242
17.2 Perspectives . . . . . 242
17.2.1 Démarche itérative entre l'extraction et la dérivation . . . . . 242
17.2.2 Amélioration de l'extraction des diagrammes d'états-transitions . . . . 243
17.2.3 Achèvement de la plate-forme JBTOOLS . . . . . 244
17.2.4 Fusion des deux méthodes Extraction/Traduction . . . . . 244
17.2.5 Validation de spécifications relativement au cahier des charges . . . . 244

**Bibliographie** **245**

# Table des figures


1.1 Cycle de développement d'un logiciel . . . . . 26

2.1 Cycle de développement en B . . . . . 33
2.2 Structure d'une machine abstraite . . . . . 34
2.3 Une machine B . . . . . 38
2.4 La machine abstraite *MachineA* . . . . . 39
2.5 Le raffinement de *MachineA* . . . . . 39
2.6 Graphe de dépendance d'un projet B . . . . . 41

4.1 Une classe pourvue d'attributs et de méthodes . . . . . 53
4.2 Généralisation . . . . . 54
4.3 Généralisation, association et multiplicité . . . . . 54
4.4 Qualification d'association . . . . . 54
4.5 Une agrégation . . . . . 55
4.6 Notation d'un état en UML . . . . . 55
4.7 Notation d'un diagramme d'états-transitions . . . . . 56
4.8 Notation d'une transition complexe . . . . . 56
4.9 Diagrammes d'états-transitions concurrents . . . . . 57
4.10 Diagrammes concurrents au sein d'un état . . . . . 57
4.11 Les acteurs d'un diagramme de séquence . . . . . 57
4.12 Les messages d'un diagramme de séquence . . . . . 58
4.13 Diagramme de séquence avec temps d'exécution des méthodes . . . . . 58
4.14 Utilisation de conditions dans un diagramme de séquence . . . . . 59
4.15 Utilisation de pseudo-codes dans un diagramme de séquence . . . . . 59
4.16 Utilisation de contraintes temporelles dans un diagramme de séquence . . . . 59
4.17 Les acteurs d'un diagramme d'activité . . . . . 60
4.18 Exécutions concurrentes de branches d'un diagramme d'activité . . . . . 60
4.19 Les points de choix dans les diagrammes d'activité . . . . . 61

5.1 Une classe en UML . . . . . 66
5.2 Dérivation de la classe en B . . . . . 66
5.3 Une classe *template* . . . . . 66
5.4 Dérivation d'une classe *template* . . . . . 66
5.5 Un attribut . . . . . 67
5.6 Dérivation d'un attribut . . . . . 67
5.7 Une méthode . . . . . 68
5.8 Dérivation d'une méthode . . . . . 68

5.9 Un constructeur . . . 68
5.10 Dérivation du constructeur . . . 69
5.11 Une association . . . 69
5.12 Dérivation d'une association . . . 69
5.13 Une généralisation . . . 70
5.14 Dérivation de la généralisation . . . 71
5.15 Des états . . . 73
5.16 Dérivation des états selon Sekerinski . . . 73
5.17 Dérivation des états selon Meyer . . . 73
5.18 Initialisation déterministe . . . 74
5.19 Initialisation non déterministe . . . 74
5.20 Dérivation de l'initialisation selon Sekerinski . . . 74
5.21 Dérivation de l'initialisation selon Meyer . . . 74
5.22 Des transitions . . . 75
5.23 Dérivation des transitions selon Sekerinski . . . 75
5.24 Dérivation d'une transition selon Meyer . . . 75
5.25 Une transition complexe . . . 76
5.26 Dérivation d'une transition complexe . . . 76
5.27 Diagramme de classes de l'ascenseur . . . 77
5.28 Diagramme d'états-transitions du bouton . . . 77
5.29 Diagramme d'états-transitions de l'ascenseur . . . 78
5.30 Spécification de la partie *system* de l'ascenseur . . . 78
5.31 Spécification de la machine *Types* de l'ascenseur . . . 79
5.32 Spécification de l'ascenseur : cas général (travaux de Meyer) . . . 80
5.33 Implémentation de la machine *system* . . . 81
5.34 Architecture de la spécification B du système de contrôle d'ascenseurs . . . 81

6.1 La machine *AM* . . . 88
6.2 Modélisation de la machine *AM* en classe . . . 88
6.3 Machine munie de variables entières et booléennes . . . 88
6.4 Modélisation des variables . . . 88
6.5 Machine munie d'ensembles abstraits . . . 89
6.6 Modélisation des variables typées avec des ensembles abstraits . . . 89
6.7 Contraintes d'inclusion dans un ensemble $A$ . . . 90
6.8 Machine munie d'ensembles énumérés . . . 90
6.9 Modélisation des ensembles énumérés et variables typées avec ceux-ci . . . 90
6.10 Contrainte relative à l'inclusion, l'appartenance et l'égalité . . . 91
6.11 Machine munie de produit cartésien . . . 91
6.12 Modélisation du produit cartésien . . . 91
6.13 Machine utilisant des relations . . . 92
6.14 Modélisation des relations . . . 92
6.15 Une autre modélisation des relations . . . 93
6.16 Diagramme de classes d'une machine avec une fonction $F$ de $X$ dans $Y$ . . . 93
6.17 Contrainte relative aux fonctions totales . . . 93
6.18 Contrainte relative aux fonctions partielles . . . 94
6.19 Contrainte relative aux injections totales . . . 94
6.20 Contrainte relative aux injections partielles . . . 94

6.21 Contrainte relative aux surjections totales . . . . . 95
6.22 Contrainte relative aux surjections partielles . . . . . 95
6.23 Contrainte relative aux bijections totales . . . . . 95
6.24 Autre écriture possible de la contrainte relative aux bijections totales . . . . . 96
6.25 Contrainte relative aux bijections partielles . . . . . 96
6.26 Exemple de la gestion des inscriptions . . . . . 97
6.27 Diagramme de classes correspondant à la machine abstraite *GestionDesInscriptions* 98
6.28 Contrainte d'inclusion ensembliste de *InscritsAuxExamens* dans *Inscrits* . . . . 98
6.29 Contrainte relative à la fonction partielle *Inscrits* . . . . . 98
6.30 Extrait de la spécification B du contrôle d'accès aux bâtiments . . . . . 100
6.31 Les trois solutions possibles en UML . . . . . 101
6.32 Notre proposition . . . . . 102

7.1 Extrait de la spécification B de l'exemple du *Scheduler* . . . . . 107
7.2 Modélisation en diagramme de classes de la machine *Scheduler* . . . . . 107
7.3 Transformation d'une propriété invariante du *Scheduler* . . . . . 108
7.4 Transformation en OCL d'une propriété invariante complexe . . . . . 108
7.5 Autre extrait de la spécification du *Scheduler* . . . . . 110
7.6 Contrainte OCL extraite de l'exemple du *Scheduler* . . . . . 111

8.1 Des variables . . . . . 115
8.2 Extraction des variables . . . . . 115
8.3 Une variable typée à l'aide d'un ensemble énuméré . . . . . 116
8.4 Extraction d'une variable typée à l'aide d'un ensemble énuméré . . . . . 116
8.5 Extraction de la substitution *Identité* . . . . . 117
8.6 Extraction des substitutions simples non déterministes . . . . . 117
8.7 Extraction de la substitution *Devient égal simple* . . . . . 117
8.8 Principe général de l'extraction . . . . . 118
8.9 La substitution *Bloc* . . . . . 118
8.10 Extraction de la substitution *Bloc* . . . . . 119
8.11 La substitution *Sélection* . . . . . 119
8.12 Extraction de la substitution *Sélection* . . . . . 119
8.13 La substitution *Précondition* . . . . . 119
8.14 Extraction de la substitution *Précondition* . . . . . 120
8.15 La substitution *Choix borné* . . . . . 120
8.16 Extraction de la substitution *Choix borné* . . . . . 120
8.17 La substitution *Condition* . . . . . 120
8.18 Extraction de la substitution *Condition* . . . . . 121
8.19 La substitution *Condition par cas* . . . . . 121
8.20 Extraction de la substitution *Condition par cas* . . . . . 122
8.21 La substitution *Choix non borné* . . . . . 122
8.22 Extraction de la substitution *Choix non borné* . . . . . 122
8.23 Extraction de la substitution *Sélection* après interprétation de $P$ . . . . . 123
8.24 Deux variables concurrentes . . . . . 124
8.25 Traduction des variables concurrentes . . . . . 124
8.26 Une opération avec paramètre . . . . . 124
8.27 Traduction des paramètres . . . . . 124

8.28 Imbrication de substitutions de composition . . . . . 125
8.29 Exemple de machine . . . . . 126
8.30 Automate des variables *v1* et *v2* . . . . . 127
8.31 Automate final de la machine *M1* . . . . . 127
8.32 Le robot . . . . . 128
8.33 Machine abstraite . . . . . 128
8.34 Diagramme d'états transitions du *Robot0* extrait de l'initialisation . . . . . 129
8.35 Diagramme d'états transitions du *Robot0* extrait de l'opération *Chgt* . . . . . 129
8.36 Diagramme extrait de l'opération *Chgt* après simplification . . . . . 129
8.37 Diagramme extrait de l'opération *Dchgt* après simplification . . . . . 130
8.38 Diagramme d'états-transitions du *Robot0* . . . . . 130
8.39 Premier raffinement du robot . . . . . 131
8.40 Diagramme d'états-transitions de la variable *Type* pour *Chgt* . . . . . 131
8.41 Diagramme d'états-transitions de la variable *Type* pour *Chgt* après simplification 132
8.42 Diagramme d'états-transitions complet de la variable *Type* . . . . . 132
8.43 Diagramme d'états-transitions des variables *Dt1, De1, De2* et *Da1* . . . . . 132
8.44 Diagramme d'états-transitions $D_{Dt1,Type}$ . . . . . 133
8.45 Diagramme d'états-transitions $D_{Dt1,Type,Da1,De1,De2}$ . . . . . 134
8.46 La machine *Parking* . . . . . 135
8.47 Système de transitions étiquetées relatif à la machine *Parking* . . . . . 135
8.48 Le raffinement de la machine *Parking* . . . . . 136
8.49 Système de transitions étiquetées relatif au raffinement *Parking_r1* . . . . . 136
8.50 Diagramme d'états-transitions relatif à la machine abstraite *Parking* . . . . . 137
8.51 Diagramme d'états-transitions des variables du raffinement *Parking_r1* . . . . . 137
8.52 Diagramme d'états-transitions $D_{cc,feu}$ du raffinement *Parking_r1* . . . . . 138

9.1 Une implémentation . . . . . 140
9.2 La traduction d'une implémentation en UML . . . . . 140
9.3 Une opération en B0 . . . . . 140
9.4 La traduction de l'appel d'une opération . . . . . 141
9.5 Une opération qui retourne des valeurs . . . . . 141
9.6 Traduction d'une opération qui retourne des valeurs . . . . . 141
9.7 Instructions IF et CASE . . . . . 142
9.8 Traduction des appels d'opération à l'intérieur d'une instruction conditionnelle 142
9.9 Une boucle en B0 . . . . . 143
9.10 Traduction d'une boucle . . . . . 143
9.11 Liens de visibilité et importation entre machines . . . . . 144
9.12 Traduction des liens de visibilité et importation entre machines . . . . . 144
9.13 Initialisation et opérations dans une machine B . . . . . 144
9.14 Traduction de l'initialisation et des opérations . . . . . 145
9.15 Le séquencement . . . . . 145
9.16 Traduction du séquencement . . . . . 145
9.17 Instructions conditionnelles simples . . . . . 146
9.18 Traduction des expressions conditionnelles simples en UML . . . . . 146
9.19 Expressions conditionnelles complexes . . . . . 147
9.20 Traduction des expressions conditionnelles complexes en UML . . . . . 147
9.21 Expressions booléennes . . . . . 147

9.22 Une boucle en B0 . . . . . 148
9.23 Traduction d'une boucle . . . . . 148
9.24 Appel d'opération en B . . . . . 149
9.25 Traduction de l'appel d'opération . . . . . 149

11.1 Exemple du *BWallet* (Spécification B : fig 13.2) traduit à l'aide du traducteur C de l'AtelierB . . . . . 159
11.2 Objectifs du projet BOM . . . . . 160
11.3 Traduction d'une relation . . . . . 162
11.4 Un invariant VDM++ . . . . . 163
11.5 Traduction de l'invariant VDM++ en une fonction C++ . . . . . 163

12.1 Cycle de développement d'un projet B . . . . . 165
12.2 Machine abstraite et implémentation . . . . . 166
12.3 Importation nominative ou non d'une machine . . . . . 168
12.4 Création de deux instances de *M1* l'une n'étant pas nommée . . . . . 169
12.5 Utilisation de machines importées une seule fois . . . . . 170
12.6 Utilisation de deux machines (*M1* et *M2*) non instanciées dans *M3* . . . . . 171
12.7 Exemple de machine avec des paramètres ensemblistes . . . . . 172
12.8 Exemple d'importation de machines avec des paramètres ensemblistes . . . . 172
12.9 Exemple de machine avec un ensemble abstrait . . . . . 173
12.10 Exemples d'implémentation avec typage d'un ensemble abstrait . . . . . 173
12.11 Un ensemble abstrait en B . . . . . 175
12.12 Traduction d'un ensemble abstrait en Java . . . . . 176
12.13 Un ensemble énuméré en B . . . . . 176
12.14 Traduction d'un ensemble énuméré en Java . . . . . 176
12.15 Une constante en B . . . . . 177
12.16 Traduction d'une constante en Java . . . . . 177
12.17 Des variables en B . . . . . 178
12.18 Traduction des variables en Java . . . . . 178
12.19 Constantes tableaux à une dimension en B . . . . . 179
12.20 Traduction de constantes de type tableaux en Java . . . . . 179
12.21 Variables de type tableaux en B . . . . . 179
12.22 Traduction des variables de type tableaux en Java . . . . . 180
12.23 Tableaux dont les indices sont des énumérés ou des booléens en B . . . . . 180
12.24 Traduction des tableaux dont les indices sont des énumérés ou des booléens B en Java . . . . . 181
12.25 Tableau qui représente une variable ensembliste . . . . . 181
12.26 Tableaux de tableaux en B . . . . . 182
12.27 Traduction en Java de tableaux de tableaux . . . . . 182
12.28 Création de plusieurs instances paramétrées en B . . . . . 183
12.29 Traduction de plusieurs instances paramétrées en Java . . . . . 183
12.30 Un exemple de passage de paramètres en B . . . . . 185
12.31 Traduction du passage de paramètres en Java . . . . . 186
12.32 Appels d'opération interdits . . . . . 187
12.33 Un exemple de passage d'un paramètre tableau par référence . . . . . 187
12.34 Traduction en Java d'un paramètre tableau passé par référence . . . . . 188

12.35 Substitution *Devient égal* pour les tableaux . . . . . 190
12.36 Traduction en Java de la substitution *Devient égal* pour les tableaux . . . . . 191
12.37 La substitution appel d'opération en B . . . . . 192
12.38 Traduction de la substitution appel d'opération en Java . . . . . 193
12.39 Le séquencement B . . . . . 193
12.40 Traduction en Java du *Séquencement* B . . . . . 193
12.41 Les substitutions *Variables locales* en B . . . . . 193
12.42 Traduction des substitutions *Variables locales* B en Java . . . . . 194
12.43 La substitution *Condition par cas* en B . . . . . 195
12.44 Traduction en Java de la substitution *Condition par cas* de B . . . . . 196
12.45 La substitution *Boucle tant que* en B . . . . . 196
12.46 Traduction de la substitution *Boucle tant que* en Java . . . . . 196

13.1 Développement d'une applet JavaCard . . . . . 199
13.2 Spécification abstraite B du porte-monnaie électronique . . . . . 200
13.3 Implémentation B du porte-monnaie électronique . . . . . 201
13.4 La spécification B de l'exemple du porte-monnaie . . . . . 202
13.5 Traduction de l'applet *BWallet* . . . . . 203
13.6 Code Java généré pour l'exemple du porte-monnaie : contenu du porte-monnaie 204
13.7 Code Java généré pour l'exemple du porte-monnaie : point d'entrée du code Java 206
13.8 Le porte-monnaie électronique pour JavaCard écrit manuellement . . . . . 207
13.9 Implémentation JavaCard de la classe *Wallet* . . . . . 208

15.1 Architecture de la plate-forme JBTOOLS . . . . . 214
15.2 Début de la grammaire XML . . . . . 216
15.3 Traduction du typage en XML . . . . . 216
15.4 Une partie de la grammaire des fichiers XML . . . . . 217
15.5 Exemple de classes Java de la BOL correspondant aux substitutions *parallèles* 217
15.6 Autre exemple de classes Java de la BOL correspondant aux substitutions *préconditions* . . . . . 218
15.7 Le plugin pour jEdit . . . . . 219
15.8 Architecture des différents outils . . . . . 219
15.9 Diagramme de classes de la gestion des inscriptions . . . . . 221

16.1 Graphe de dépendance du projet de carrefour . . . . . 223
16.2 Spécification abstraite du composant *Feu* . . . . . 224
16.3 Implémentation du composant *Feu* . . . . . 225
16.4 Spécification abstraite du composant *CARREFOUR* . . . . . 225
16.5 Implémentation du composant *CARREFOUR* . . . . . 226
16.6 Le composant principal . . . . . 227
16.7 Diagramme de classes extrait de la spécification *CARREFOUR* . . . . . 228
16.8 Contrainte relative à la bijection totale *suiv* . . . . . 228
16.9 Diagramme de classes extrait de la spécification *Feu* . . . . . 228
16.10 Contrainte relative aux propriétés complexes de l'invariant de la spécification *CARREFOUR* . . . . . 229
16.11 Contrainte relative aux propriétés complexes de l'invariant de la spécification *Feu* . . . . . 229

16.12Extraction de la post-condition de l'opération *MiseEnService* . . . . . 229
16.13Extraction de la post-condition de l'opération *ChangerFeux* . . . . . 230
16.14Diagramme d'états-transitions extrait de la spécification *Feu* . . . . . 231
16.15Diagramme d'états-transitions extrait de la spécification *CARREFOUR* . . . . 231
16.16Diagramme de séquence extrait de la spécification *CARREFOUR_imp* . . . . 232
16.17Diagramme d'activités extrait de la spécification *MAIN_imp* . . . . . 233
16.18Traduction du composant principal en Java . . . . . 234
16.19Traduction du composant *CARREFOUR* en Java . . . . . 235
16.20Traduction du composant *Feu* en Java . . . . . 236

# Liste des tableaux


1.1 Convention d'écriture des concepts et objets des différents langages . . . . . 29
1.2 Eléments de grammaire des langages utilisés dans ce document . . . . . 29

2.1 Visibilité à l'intérieur d'une machine abstraite de ses propres entités . . . . . 35
2.2 Substitutions de base de la méthode B et leur sémantique . . . . . 35
2.3 Substitutions de composition et leur sémantique . . . . . 36
2.4 Notation étendue des substitutions du langage B . . . . . 36
2.5 Typage des expressions B . . . . . 37
2.6 Type des ensembles prédéfinis . . . . . 37
2.7 Type de visibilité . . . . . 41
2.8 Visibilité des entités d'une machine vue . . . . . 42
2.9 Visibilité des entités d'une machine incluse ou étendue . . . . . 42
2.10 Visibilité d'une machine utilisée . . . . . 42
2.11 Visibilité des entités d'une machine importée . . . . . 43

5.1 Affinement de la dérivation d'une association par ses multiplicités . . . . . 70

6.1 Stéréotypes d'aide à la lecture des diagrammes de classes . . . . . 96

7.1 Equivalence en OCL des prédicats B . . . . . 104
7.2 Equivalence en OCL des expressions arithmétiques B . . . . . 105
7.3 Equivalence en OCL des expressions ensemblistes . . . . . 105
7.4 Modélisation en OCL des expressions B portant sur les relations . . . . . 106
7.5 Equivalence en OCL des expressions sur les suites . . . . . 106

8.1 Extraction de la substitution *Condition* à l'aide de *Sélection* . . . . . 120
8.2 Extraction de la substitution *Condition par cas* à l'aide de *Sélection* . . . . . 121
8.3 Valeurs des variables pour les états de $D_{Dt1,Type}$ . . . . . 133
8.4 Valeurs des variables pour les états de $D_{Dt1,Type,Da1,De1,De2}$ . . . . . 134
8.5 Valeurs des variables pour les états de $D_{cc,feu}$ . . . . . 138

11.1 Traduction des structures de contrôle B0 en OCaml . . . . . 161

12.1 Type des entités de la machine $A$ après vérification locale des types . . . . . 170
12.2 Type des entités de la machine $A$ après vérification locale partielle des types de $B$ . . . . . 171
12.3 Type des entités de la machine $A$ après vérification locale complète des types de $B$ . . . . . 172
12.4 Proposition d'équivalence non retenue entre les types de base B et les types Java 174

12.5 Traduction des structures de contrôle B0 en Java . . . . . 189
12.6 Substitution *Condition* en B (à gauche) et en Java (à droite) . . . . . 194

# Première partie

# Présentation

# Chapitre 1

# Introduction

Au cours de ces dernières années, l'utilisation de l'informatique s'est répandue dans les applications industrielles les plus diverses. En conséquence, le développement d'applications sûres est devenu un besoin crucial au moins pour trois raisons :

- La première est le risque humain et concerne les domaines critiques tels que l'aéronautique, le ferroviaire, la défense, le nucléaire, etc...
- La seconde est la demande croissante d'applications sûres dans les domaines non critiques. Par exemple le retour de produits vendus pour une correction du logiciel embarqué ou non est très onéreux sans parler de la remise en cause de l'image de marque de la société productrice.
- La troisième est l'augmentation croissante de la complexité des logiciels avec le temps.

Cette complexité et le besoin de sécurité ont amené les industriels à utiliser des technologies de plus en plus formalisées pour concevoir leurs logiciels. Ces techniques sont souvent associés à des normes concernant le développement qui ont pour but d'obtenir certains niveaux de qualité.

On caractérise le développement logiciel par l'ensemble des techniques qui sont utilisées dans le processus complet de création c'est-à-dire de l'identification du besoin à l'utilisation effective de l'application. Les différentes activités de ce processus sont divisées en quatre phases majeures (fig 1.1) qui sont détaillées dans [26, 89]. Après l'identification du besoin d'un nouveau logiciel, il est nécessaire de connaître le cadre d'application de celui-ci. Une première phase d'analyse des besoins répond à cette question. Cette étude se base sur le domaine d'application et l'environnement du futur logiciel. Les méthodes associées à cette phase ne relèvent pas directement des techniques informatiques mais des sciences cognitives. Il en résulte une spécification informelle que l'on a coutume d'appeler le cahier des charges. A partir de celui-ci, une phase de spécification/conception permet de définir schématiquement ce à quoi le logiciel doit répondre. Ceci donne une vue d'ensemble du produit sans les détails de la réalisation. Cette phase est suivie aussitôt d'une validation qui permet de vérifier si le logiciel répond totalement aux besoins des futurs utilisateurs. Il est alors possible d'entamer l'implémentation selon la spécification qui est le développement proprement dit. Dans cette activité vont apparaître les premiers détails du logiciel. Cette phase met l'accent sur le codage de l'application. Elle consiste à mettre en place un ensemble de programmes exécutables. C'est lors de cette étape que le projet informatique commence à se matérialiser. Une fois le code du logiciel écrit une première phase de tests unitaires est nécessaire dans le but de s'assurer de son bon fonctionnement hors environnement. L'intégration dans l'environnement est une phase

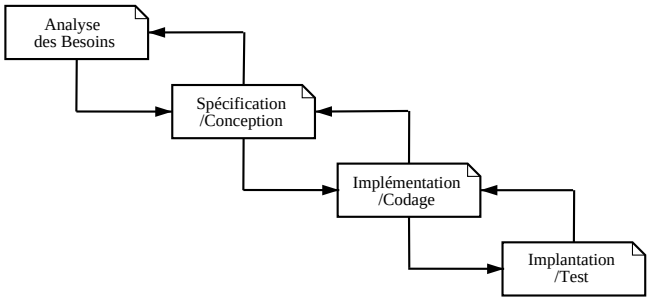


FIG. 1.1 – Cycle de développement d'un logiciel

critique. En effet les problèmes rencontrés lors de cette étape remettent en cause toutes les activités du développement. Plus généralement les erreurs rencontrées à chaque étape peuvent remettre en cause une à plusieurs, voire toutes les étapes qui la précèdent d'où l'intérêt de l'automatisation du maximum d'activités dans le cycle de développement.

Les méthodes utilisées pour guider le développement d'applications peuvent être classées selon le niveau de formalisation de leurs concepts en trois grandes familles :

- Les méthodes informelles
- Les méthodes semi-formelles
- Les méthodes formelles

L'utilisation des méthodes informelles consiste à mettre en place des normes de programmation et de certifications du code produit. Ce processus de développement a permis de développer des applications sûres mais souffre néanmoins d'un manque de vérification formelle de l'application produite. L'utilisation des méthodes semi-formelles augmenta le niveau de structuration du logiciel obtenu car elles donnent une vue graphique et synthétique du système. L'émergence des méthodes formelles permit la création d'applications totalement sûres ou presque du fait de leur précision et de l'utilisation de la preuve. Les inconvénients de l'utilisation des méthodes formelles sont le temps d'apprentissage nécessaire pour les maîtriser et le temps de développement associé. On conçoit volontiers que la disponibilité d'une méthode de modélisation unique, adaptée à toutes les situations n'existe pas.

## 1.1 Contexte

Les besoins du monde industriel sont les suivants : une augmentation de la sûreté de leurs produits et un gain de productivité au niveau de la conception. Les méthodes les plus utilisées en modélisation (Merise, OMT, UML ...), dites semi-formelles, sont basées principalement sur l'utilisation de divers diagrammes. Elles représentent le système d'une manière à la fois intuitive et synthétique. De ce fait, elles sont bien adaptées à la plupart des utilisateurs et concepteurs des systèmes d'informations. Ces avantages ont contribué à répandre très largement leur utilisation dans l'industrie. Leur principal défaut est le manque de précision de leur sémantique qui rend leur notation parfois ambiguë. De plus, ces méthodes ne permettent pas de s'assurer du bon fonctionnement du système. En effet l'expérience des analystes concepteurs est le seul parachute contre les problèmes d'exécution du système car il est impossible de prouver son fonctionnement par des moyens calculatoires. Il est donc nécessaire de mettre en place une grosse batterie de tests pour s'assurer du bon fonctionnement de la majorité des cas d'utilisation sans atteindre une sûreté complète.

Les méthodes formelles ont des caractéristiques souvent à l'opposé des méthodes semi-formelles. Elles possèdent des notations rigoureuses issues des mathématiques. Ceci permet de décrire les propriétés du système à construire de manière très précise. De ce fait ces méthodes disposent, pour certaines comme la méthode B, de techniques de preuves introduites par Hoare [51] permettant de vérifier complètement le cycle de développement des logiciels depuis les premières abstractions jusqu'à l'implantation (code compilable donc exécutable). Ces avantages sont contrebalancés par l'utilisation d'un formalisme rebutant pour beaucoup de spécialistes [43, 85] du domaine et d'un manque d'aura dans le monde industriel pour des raisons de coût et d'investissement non compressible dès le début du projet de création. En effet, le temps imparti au développement à l'aide des méthodes formelles sans aucune preuve du système est déjà supérieur à celui nécessaire pour le même développement en utilisant les méthodes semi-formelles. De plus la preuve du système augmentera encore la différence. Pourtant les méthodes formelles ont un énorme avantage sur les méthodes semi-formelles : elles permettent de réduire les tests unitaires et donc de réduire les retours aux étapes antérieures dans le cycle de vie d'un logiciel. Par conséquent elles peuvent introduire un gain de temps et donc d'argent sur la fin du développement. De par leur complémentarité l'intégration d'une méthode formelle et d'une méthode semi-formelle peut apporter une amélioration notable.

## 1.2 Choix de la méthode formelle

Dans notre contexte nous avons choisi la méthode formelle B pour diverses raisons détaillées dans les lignes suivantes. De nombreux travaux sont effectués pour et autour de B dans une communauté active [1]. Une évaluation quantitative a été réalisée par Mariano [71]. Cette évaluation est élaborée par rapport à certaines caractéristiques du développement formel comme la taille, la complexité ou la structure. Elle constitue une garantie que les spécifications sont bien formées. Cette méthode possède des fondements théoriques solides définis dans le B-Book [3]. Elle est bien outillée. Il existe actuellement deux outils autour de B qui sont l'AtelierB [92] et le BToolKit [15]. Les services offerts par ces outils sont les suivants :

- Analyse lexicale et syntaxique des modèles
- Vérification du typage
- Génération des obligations de preuves
- Prouveur automatique et interactif
- Traduction vers des langages cibles (C++, ADA, C, etc)

Ils permettent l'utilisation de la méthode dans le cycle de vie des projets industriels tels que le dispositif de contrôle de vitesse du métro de Calcutta [17], le projet METEOR [37, 6] (construction de la première ligne automatisée du métro parisien - Ligne 14).

## 1.3 Contributions et objectifs de recherche

La contribution de nos travaux se situe à deux instants clefs du cycle de développement d'un logiciel (fig 1.1) :

1. Le premier est la validation du cahier des charges à la fin de l'étape de spécification/conception. En effet avant de passer à l'étape suivante, l'implantation, il est nécessaire de vérifier la concordance entre les spécifications des concepteurs et le cahier des charges

[1] http ://www-lsr.imag.fr/B/

définis par les utilisateurs du logiciel. La démarche consiste à extraire des vues semi-formelles depuis la spécification B. De cette manière on obtient une vue d'ensemble du projet bien plus synthétique que les spécifications brutes écrites en B. Il est ensuite relativement simple pour un spécialiste d'UML de valider les vues UML par rapport au cahier des charges. Il faut noter que lors de cette étape le spécialiste ne créera pas un modèle directement en UML pour comparer avec le modèle extrait. En effet, il n'existe pas une unique manière d'écrire un modèle UML et par conséquent le modèle extrait pourrait être différent de celui réalisé par l'expert. Le spécialiste vérifiera la conformité entre le modèle et les règles de gestion du cahier des charges. Les gains de productivité demandés dans l'industrie nous ont poussé à penser cette étude dans l'optique d'une automatisation.

2. Le deuxième est la traduction de spécifications B vers les langages à objets. Ceci est un des résultats du projet RNTL[2] BOM[3]. Le but était de fournir des traducteurs optimisant pour le langage formel B, ces traducteurs devant permettre une utilisation de la méthode B, traduite pour des plates-formes fortement contraintes de type carte à puce. Notre traducteur devait être Open-Source, dans le but de permettre une diffusion des outils auprès de la communauté B. De plus il devait être optimisé en termes d'occupation mémoire et permettre la réalisation en B d'applications prouvées pour les JavaCards. Pour cela, un traducteur optimisant a été développé depuis le langage de spécification B vers les langages à objets en tenant compte des restrictions de la plate-forme cible.

## 1.4 Structure du document et guide de lecture

Après une introduction et la présentation de la méthode B qui est au cœur de nos travaux, ce travail de thèse s'inscrit, pour la partie 1, dans l'étude de deux techniques de spécification : la méthode B et la notation UML. Il s'appuie sur la complémentarité de ces deux approches et contribue à l'utilisation conjointe des langages formels (B) et des notations graphiques à objets (UML). Cette partie est composée des présentations d'UML et d'un état de l'art des travaux sur l'intégration de ces deux méthodes de spécification. Ensuite nous présentons notre contribution à cette intégration selon deux parties statique et dynamique en cinq chapitres qui sont :

1. Premièrement l'extraction des données statiques de la spécification :
   - Diagrammes de classes
   - Génération de contraintes OCL
2. Deuxièmement l'extraction des données dynamiques de la spécification :
   - Diagrammes d'états-transitions
   - Diagrammes d'activité
   - Diagramme de séquence

D'autre part nous exposons, en partie 2, la génération automatique de code écrit dans un langage à objets à partir de spécifications B. Cette partie 2 commence par un bref état de l'art des différentes techniques déjà mises en œuvre suivies de notre contribution dans le

[2]Réseau National des Technologies Logicielles : Le Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL) a été mis en place, dès 1999, par le ministère de la Recherche et le ministère de l'Industrie. Il a reçu comme mission de favoriser la constitution de projets innovants de recherche et de développement coopératif entre entreprises et équipes de la recherche publique.

[3]http ://lifc.univ-fcomte.fr/~tatibouet/WEBBOM

domaine dont le principe novateur est la plate-forme cible : la JavaCard. Une évaluation de cet outil conclura cette partie. Ensuite une troisième partie conclura ces travaux de recherche par la présentation des outils déjà développés et la mise en œuvre dans un cas d'étude. Enfin nous conclurons sur les points forts, les points faibles, les améliorations et perspectives de cette étude.

### 1.4.1 Guide de lecture

Ce document de thèse concerne plusieurs langages, nous avons utilisé une convention d'écriture pour reconnaître les concepts et objets de ces différents langages. Cette convention est décrite dans le tableau 1.1 où la chaîne *expression* correspond aux fonts distinctives.

| Langages | Concepts | Objets |
|---|---|---|
| B/VDM | EXPRESSION | *expression* |
| UML/OCL | **expression** | *expression* |
| Java/OCaml/C/C++ | **expression** | *expression* |

TAB. 1.1 – Convention d'écriture des concepts et objets des différents langages

Les spécifications utilisées comme exemple dans les différentes traductions vers UML ou les langages à objets sont écrits à l'aide de règles de grammaire décrites dans le tableau 1.2.

| Eléments | Sémantique |
|---|---|
| [A] | Les crochets signifient que l'entité A est optionnelle |
| A+ | Le signe + signifie que l'élément A apparaît au moins une fois |
| A* | Le signe * signifie que l'élément A apparaît de zéro à un nombre fini de fois |
| $A^n$ | Le signe $^n$ signifie que l'élément A apparaît n fois |
| A\|B | Le caractère \| signifie que soit l'expression A<br>soit l'expression B apparaissent mais pas les deux en même temps |

TAB. 1.2 – Eléments de grammaire des langages utilisés dans ce document

### 1.4.2 Publications issues de ce travail de thèse

1. B. Tatibouet, J.C. Voisinet, "jBTools and B2UML : a platform and a tool to provide a UML Class Diagram since a B specification". ICSSEA'2001, 14th International Conference on Software & Systems Engineering and Their Applications, Vol. 2, Session 8-3, Formal Methods, France, Paris, 4-6 December 2001.
2. A. Hammad, B. Tatibouet, J.-C. Voisinet, "De la spécification B vers les diagrammes d'états-transitions UML", 7ème MCSEAI'2002 (Maghrebian Conference of Software Engineering and Intelligence Artificial), pages 133-143, Annaba, Algérie, 6-8 mai 2002.
3. J.C. Voisinet, B. Tatibouet, A. Hammad, "jBTools : An experimental platform for the formal B method", Principles and Practice of Programming in Java (PPPJ'02), p. 137-140, Trinity College, Dublin, Ireland, June 13-14 2002.
4. A. Hammad, B. Tatibouet, J.-C. Voisinet, Wu Weiping, "From a B Specification to UML Statechart Diagrams", 4th International Conference on Formal Engineering Methods (ICFEM'2002), LNCS 2495, p. 511-522, Shanghaï, China, 21-25 October 2002.

5. B. Tatibouet, A. Hammad, J.-C. Voisinet, "From an abstract B specification to UML class diagrams". 2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002), p. 5-10, ISBN 0-9727186-0-5, Marrakech, Maroc, 18-21 December 2002.

6. B. Tatibouet, A. Requet, J.C. Voisinet, A. Hammad, "JavaCard Code Generation from B Specifications", 5th International Conference on Formal Engineering Methods (ICFEM'2003), LNCS 2885, p. 306-318, Singapore, November 5-7 2003.

7. B. Tatibouet, J.C. Voisinet, "Generating statecharts from B specifications", 16th International Conference on Software & Systems Engineering and their applications (ICSSEA'2003), Vol 1, Session 2 : Modeling, December 2-4 2003, CNAM - Paris, France.

# Chapitre 2

# La méthode B

## 2.1 Introduction

Les méthodes formelles permettent d'écrire des spécifications rigoureuses, précises et complètes, et de développer du logiciel à partir de celles-ci. Leur utilisation permet le développement de produits sûrs ce qui est recherché dans les domaines critiques [5] tels que les applications bancaires, aéronautiques, etc. Dans ce chapitre nous présentons brièvement les différents types de méthodes formelles et leur environnement. Nous nous concentrerons ensuite sur la méthode utilisée dans cette étude qui est la méthode B. Les aspects de cette méthode présentés sont :

- Syntaxe
- Sémantique
- Preuves
- Mécanismes de composition

## 2.2 Les méthodes formelles

Les méthodes formelles [29] sont une solution pour concevoir des logiciels fiables. Leur syntaxe et leur sémantique sont claires, précises et accompagnées de règles de déduction conduisant à la preuve du fonctionnement du système. Il existe plusieurs types de méthodes formelles comme par exemple :

1. Les langages de spécification algébrique (orientés propriétés) : ces langages sont basés sur la notion de type abstrait algébrique. La présentation d'un type abstrait algébrique est constituée d'une partie déclaration (signature des opérations du type) et d'une partie axiomatique (équations conditionnelles ou prédicats du premier ordre sur les termes construits à partir de ces opérations). Citons par exemple :
   - LARCH [46]
   - OBJ [44]
   - CASL [9]
2. Les langages de spécification ensembliste (orientés modèles). Ils sont basés sur la théorie des ensembles et des prédicats du premier ordre. Ils utilisent des types de données prédéfinis pour modéliser l'état du système à construire. Chaque opération est spécifiée indépendamment en décrivant son effet sur l'état du système. Voici quelques exemples de ce type de langage :

- Z [90] et ObjectZ [20]
- B [3]
- VDM [58] et VDM++ [24]

Le point fort des méthodes formelles est la création de logiciels sûrs et fiables par **vérification formelle**. Elle intervient après la création du modèle mathématique d'un système et l'utilisation d'un langage pour spécifier les propriétés du système. Elle est basée sur une preuve mathématique (utilisant axiomes et règles d'inférence). Elle consiste à décrire le système et les propriétés auxquelles il doit obéir. Ensuite on démontre que les propriétés peuvent être obtenues à partir du système à l'aide d'un prouveur comme :

- PVS [78]
- HOL
- Coq [14]

Les prouveurs peuvent être inclus dans un outil comme l'AtelierB [91] ou le B-ToolKit [15] pour la méthode B.

## 2.3 La méthode B

Nous nous intéressons à la méthode B. Ses fondements théoriques ont été définis dans le B-Book [3]. Pour résumer elle utilise les prédicats du premier ordre avec la théorie des ensembles ainsi que le langage des substitutions généralisées. Le cycle de développement des applications (fig 2.1) passe par la rédaction du cahier des charges et sa spécification abstraite [60]. Par la suite les caractéristiques du cycle de développement avec B sont utilisées. La spécification abstraite est prouvée puis raffinée tout en préservant les propriétés prouvées dans l'entité abstraite. Pour finir les implémentations du projet B sont traduites vers un langage compilable et exécutable.

Il existe deux approches autour de la méthode B. L'approche événementielle, dite aussi système [1, 4], utilise les événements pour modifier l'état du système. Ils peuvent s'exécuter si leur garde (condition) est vraie. L'approche logicielle dite aussi classique utilise des opérations (équivalent des événements de la forme événementielle) qui ont des pré-conditions (en lieu et place des gardes des événements de la forme événementielle). Celles-ci doivent être vérifiées avant l'appel par nom des opérations. Dans ce document nous nous concentrerons sur la présentation de l'approche logicielle dite classique.

### 2.3.1 Fondements de la méthode B

La méthode B ne définit pas une application mais un modèle mathématique. Ce modèle est décomposé en modules (machines abstraites) pour cloisonner les contraintes définies dans le cahier des charges. De cette manière on réduit la complexité en évitant les interactions entre les différentes contraintes et leur résolution. L'élément essentiel de la méthode B est la machine abstraite (fig 2.2) qui définit le niveau le plus abstrait du système spécifié. Chaque machine abstraite pour définir un module du système est divisé en deux :

- Une partie statique qui comprend les données (ensembles, relations, fonctions, variables etc) et l'invariant qui définit les contraintes que le système doit absolument respecter.
- Une partie dynamique permet de modifier les variables grâce à l'initialisation et aux opérations. En effet l'état du système est défini par les valeurs de ces variables et ce sont les opérations qui le font vivre et évoluer dans son environnement.

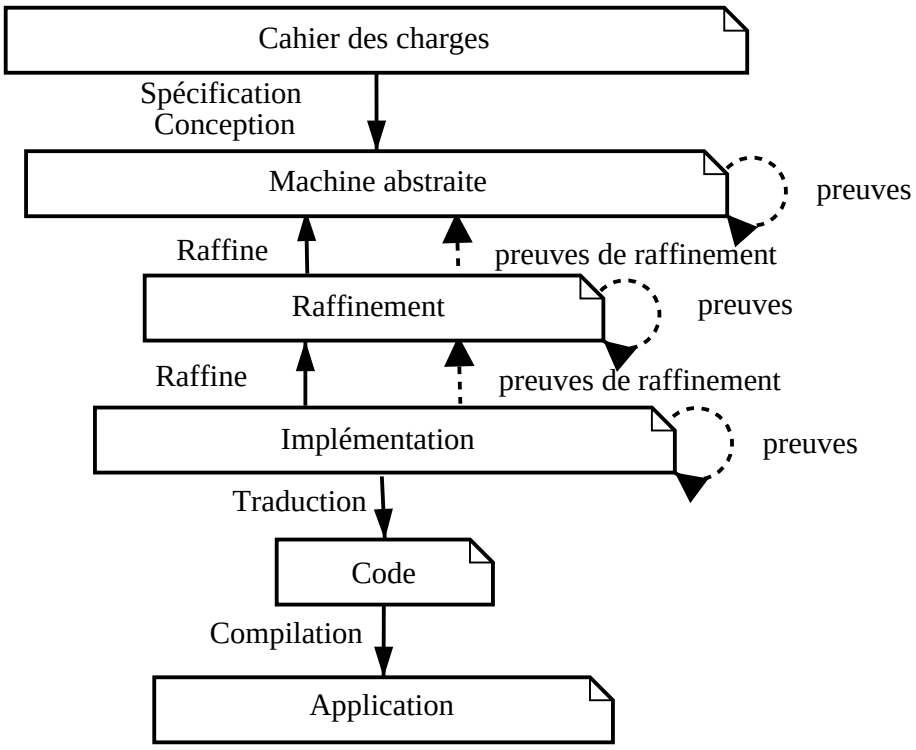


FIG. 2.1 – Cycle de développement en B

### 2.3.2 Raffinement et implémentation

Le cycle de développement de logiciels avec la méthode B commence par une abstraction. Cette abstraction définit les grandes lignes et principes essentiels du fonctionnement du système. Les raffinements successifs permettent de détailler ce fonctionnement pour le rendre de plus en plus concret jusqu'à en décrire une implémentation à l'aide du langage B0. Le langage B0 est un sous-ensemble du langage B. Il possède une syntaxe et une sémantique très proche des langages de programmation traditionnels ce qui permet une traduction aisée vers ceux-ci. Le principe de concrétisation utilisé dans le développement permet de scinder la complexité des problèmes. Durant le raffinement, l'abstraction sera détaillée. Ces détails rendent plus précis les données et les opérations de la machine abstraite.

### 2.3.3 Preuve

Les contraintes auxquelles doit obéir le système sont définies par les règles de gestion du cahier des charges. Elles sont décrites en B dans l'invariant sur l'état du système sous une forme unique de propriété statique : elles définissent des contraintes qui devront être vérifiées dans tous les états du système. Elles restreignent l'espace de valeurs que peuvent prendre les variables. Ces propriétés peuvent s'appliquer sur une ou plusieurs variables. On les appellera prédicat de typage lorsqu'elles sont en fonction d'une seule variable. Les opérations modifient les valeurs des variables et donc l'état du système. En conséquence la preuve va consister à vérifier la conformité de l'état du système avant et après l'opération. Le développement d'ap-

plication à l'aide de la méthode B débute par une machine abstraite comprenant un invariant comme tous les autres composants. Les premières obligations de preuves à remplir sont à ce niveau (fig 2.1). D'autre part les différents invariants des raffinements et implémentations apportent aussi éventuellement leurs propres obligations de preuves. Enfin le raffinement des modules apportera les obligations de preuves relatives à la préservation de l'invariant des abstractions dans le composant courant.

### 2.3.4 Structure générale d'une machine abstraite

L'élément essentiel de la méthode B est la machine abstraite (fig 2.2) qui définit le niveau le plus abstrait du système spécifié.

| | |
|---|---|
| MACHINE | <*Name*>(*paramètres*) |
| CONSTRAINTS | <*Constraints*> |
| SETS | <*Sets*> |
| CONSTANTS | <*Constants*> |
| PROPERTIES | <*Properties*> |
| DEFINITIONS | <*Definitions*> |
| VARIABLES | <*Variables*> |
| INVARIANT | <*Invariant*> |
| INITIALISATION | <*Initialisation*> |
| OPERATIONS | <*Operations*> |
| END | |

FIG. 2.2 – Structure d'une machine abstraite

1. La clause MACHINE définit le nom du composant et les paramètres éventuels. Ces paramètres peuvent être scalaires ou ensemblistes.
2. La clause CONSTRAINTS spécifie le typage des paramètres décrits dans la clause MACHINE.
3. La clause SETS déclare les ensembles utilisés dans le composant. Ils peuvent être de deux sortes. Le premier est de type énuméré par exemple *BOOLEAN* = {*false,true*} ou abstrait par exemple S.
4. La clause CONSTANTS déclare les constantes du composant.
5. La clause PROPERTIES permet de définir des contraintes sur des ensembles abstraits ou des constantes définies dans la clause CONSTANTS.
6. La clause DEFINITIONS déclare des alias textuels, éventuellement paramétrés, qui seront remplacés au début de l'analyse lexicale par leurs corps. Par exemple *Affect(x,v)* == *x := 2 * v + 1* permettra de remplacer *Affect(v1,v2)* par *v1 := 2 * v2 + 1*.
7. La clause VARIABLES déclare la liste des variables du composant.
8. La clause INVARIANT définit par des prédicats les contraintes relatives aux variables y compris le typage.
9. La clause INITIALISATION fixe la valeur initiale de chaque variable du composant par des substitutions.

10. La clause OPERATIONS utilise aussi les substitutions afin de faire évoluer l'état du système. Les opérations peuvent prendre des paramètres d'entrée ainsi que de retour.

Les règles de visibilité [93] des différents constituants à l'intérieur des différentes clauses de la machine sont décrites dans le tableau suivant (Tab 2.1).

| Constituants | Clauses | | | |
|---|---|---|---|---|
| | CONSTRAINTS | PROPERTIES | INVARIANT | INITIALISATION et OPERATIONS |
| Paramètres | X | | X | X |
| Ensembles | | X | X | X |
| Constantes | | X | X | X |
| Definitions | X | X | X | X |
| Variables | | | X | X |
| Opérations | | | | |

TAB. 2.1 – Visibilité à l'intérieur d'une machine abstraite de ses propres entités

### 2.3.5 Substitutions généralisées

Le langage B utilise la théorie des ensembles, la logique des prédicats du premier ordre et les substitutions généralisées. Les substitutions sont utilisées par l'initialisation et les opérations pour modifier les valeurs des variables dans le but de faire évoluer l'état du système. La notion de substitution est apparentée aux affectations des langages de programmation traditionnels. Elles ont été introduites en B dans le but de définir des transformateurs de prédicats [45] par rapport à un état *pré* avant la substitution et *post* après. Soit P la post-condition et S une substitution alors [S]P est la plus faible pré-condition ([19]) nécessaire pour que S se termine et établisse P. Dans le tableau (Tab 2.2), qui correspond aux substitutions simples de modification des valeurs des variables :

- x et y sont des variables
- S un ensemble
- P, Q et R des prédicats.

| **Description** | **Substitution** | **Sémantique** |
|---|---|---|
| Identité | *skip* | [skip]R $\Leftrightarrow$ R |
| Devient tel que | x : (P) | [x :(P)]R $\Leftrightarrow$ [@x'.[x := x']P$\Rightarrow$x := x']R |
| Devient élément de | x :$\in$ S | [x :$\in$ S]R $\Leftrightarrow$ [@x'.x' $\in$ S $\Rightarrow$x := x']R |
| Devient égal simple | x := E | [x :=E]R $\Leftrightarrow$ R[E/x] |
| Devient égal multiple | x,y := E,F | [x,y :=E,F]P $\Leftrightarrow$ P[E,F/x,y] |

TAB. 2.2 – Substitutions de base de la méthode B et leur sémantique

Dans le tableau 2.2 les substitutions de base du langage B sont présentées avec leur sémantique. Par exemple la sémantique de la substitution *Devient élément de* est [x :$\in$ S]R $\Leftrightarrow$ [$\exists$x'.x' $\in$ S $\Rightarrow$ x := x']R où [$\exists$x'.x' $\in$ S $\Rightarrow$ x := x']R dénote le remplacement d'une occurrence libre de x dans R par x' qui appartient à l'ensemble S. La substitution multiple (x,y := E,F) est un cas particulier car elle est composée à partir de la substitution simple et de la composition parallèle ||. Cette substitution peut s'exprimer sous les deux formes suivantes :

- $x,y := E,F$
- $x := E \parallel y := F$

Il existe des substitutions de composition (Tab 2.3) qui permettent de composer une substitution complexe avec des substitutions simples.

| Description | Substitution | Sémantique |
|---|---|---|
| Séquencement | G ;H | [G ;H]R $\Leftrightarrow$ [G][H]R |
| Préconditionné | P\|G | [P\|G]R $\Leftrightarrow$ P $\wedge$ [G]R |
| Gardé | P $\Rightarrow$ G | [P $\Rightarrow$ G]R $\Leftrightarrow$ P$\Rightarrow$[G]R |
| Choix borné | G [] H | [G [] H]R $\Leftrightarrow$ [G]R $\wedge$ [H]R |
| Choix non borné | @x.(G) | [$\exists$x.(G)]R $\Leftrightarrow$ $\forall$x.([G]R) |

TAB. 2.3 – Substitutions de composition et leur sémantique

| Description | Substitution | Sémantique |
|---|---|---|
| Bloc | BEGIN *G* END | G |
| Conditionnelle | IF *P* THEN *G* ELSE *H* END | (P $\Rightarrow$ G) [] ($\neg$P $\Rightarrow$ H) |
| Choix borné | CHOICE *G* OR *H* END | G [] H |
| Choix non borné | ANY *x* WHERE *P* THEN *G* END | $\forall$x(P$\Rightarrow$G) |
| Précondition | PRE *P* THEN *G* END | P\|G |
| Sélection | SELECT *P* THEN *G* ... WHEN *Q* THEN *H* END | (P$\Rightarrow$G) [] ... [] (Q$\Rightarrow$H) |
| Variable locale | VAR *x* IN *G* END | $\forall$x.(G) |
| Définition locale | LET *y* BE *y* = *E* IN *G* END | $\forall$y.(y = E $\Rightarrow$ G) |

TAB. 2.4 – Notation étendue des substitutions du langage B

### 2.3.6 Théorie des ensembles et typage

La méthode B utilise la théorie des ensembles comme base de ses modèles mathématiques. La méthode B n'est pas un langage de programmation traditionnel comme PASCAL, C, etc, car les variables ne sont pas typées à leur déclaration. En effet, elles sont typées dans la clause INVARIANT à l'aide de prédicats du premier ordre et par conséquent le typage est différent de ceux utilisés dans un langage traditionnel. La seule similitude concerne les variables scalaires qui correspondent aux variables du type entier, caractère, etc. utilisées dans les langages traditionnels (Tableau 2.6). Les types basiques utilisés par la méthode B sont les suivants :

- *BOOL* est le type booléen défini par *TRUE* et *FALSE*
- $\mathbb{Z}$ est l'ensemble des entiers relatifs défini par ]-$\infty$,+$\infty$[. Les constantes *MAXINT* et *MININT* qui correspondent à l'ensemble des entiers représentables directement sans débordement sur la machine cible représentent l'ensemble INT des entiers utilisable en B0.
- *STRING* est le type des chaînes de caractères.
- *Ident* correspond aux types des ensembles énumérés définis dans la clause SETS.

Pour décrire les types on utilise les notations suivantes :

- $\mathbb{P}(T)$ désigne l'ensemble des parties de $T$, c'est-à-dire l'ensemble dont les éléments sont des ensembles d'éléments de $T$ sachant que $T$ représente un type quelconque.

– $T \times U$ désigne le produit cartésien des ensembles $T$ et $V$, c'est-à-dire l'ensemble des paires ordonnées dont le premier élément est de type $T$ et le second de type $V$. L'opérateur est associatif à gauche, le type T × U × V × W désigne en réalité le type ((T × U) × V) × W.

| **Description** | **Notation** | **Type de Exp1** | **Type de Exp2** | **Type** |
|---|---|---|---|---|
| Séquence | *seq(Exp1)* | $\mathbb{P}$(T1) | | $\mathbb{P}(\mathbb{P}(\mathbb{Z} \times$ T1)) |
| Relation | *Exp1* $\leftrightarrow$ *Exp2* | $\mathbb{P}$(T1) | $\mathbb{P}$(T2) | $\mathbb{P}(\mathbb{P}($T1 × T2)) |
| Fonction | *Exp1* $\rightarrow$ *Exp2* | $\mathbb{P}$(T1) | $\mathbb{P}$(T2) | $\mathbb{P}(\mathbb{P}($T1 × T2)) |
| Bijection | *Exp1* $\rightarrowtail$ *Exp2* | $\mathbb{P}$(T1) | $\mathbb{P}$(T2) | $\mathbb{P}(\mathbb{P}($T1 × T2)) |
| Injection | *Exp1* $\rightarrowtail$ *Exp2* | $\mathbb{P}$(T1) | $\mathbb{P}$(T2) | $\mathbb{P}(\mathbb{P}($T1 × T2)) |
| Surjection | *Exp1* $\twoheadrightarrow$ *Exp2* | $\mathbb{P}$(T1) | $\mathbb{P}$(T2) | $\mathbb{P}(\mathbb{P}($T1 × T2)) |
| Domaine | *dom(Exp1)* | $\mathbb{P}$(T1 × T2) | | $\mathbb{P}$(T1) |
| Codomaine | *ran(Exp1)* | $\mathbb{P}$(T1 × T2) | | $\mathbb{P}$(T2) |
| Produit direct | *Exp1* $\otimes$ *Exp2* | $\mathbb{P}$(T1 × T2) | $\mathbb{P}$(T1 × T3) | $\mathbb{P}$(T1 × (T2 × T3)) |
| Composition | *Exp1 ; Exp2* | $\mathbb{P}$(T1 × T2) | $\mathbb{P}$(T2 × T3) | $\mathbb{P}$(T1 × T3) |
| Inverse | $Exp1^{-1}$ | $\mathbb{P}$(T1 × T2) | | $\mathbb{P}$(T2 × T1) |

TAB. 2.5 – Typage des expressions B

D'autre part il existe deux opérateurs de typage dans la méthode B qui sont $\in$, $\subset$, $=$ et $:=$. Si une variable *var* est typée de la manière suivante : *var* $\in E$ avec $E$ une expression de type $\mathbb{P}$(T) alors le type de *var* est T. On parle de variable scalaire lorsque le type de la variable n'est pas constitué d'un produit cartésien ou de partie d'un type quelconque. Si une variable *var* est typée de la manière suivante : *var* $\subset E$ avec $E$ une expression de type $\mathbb{P}$(T) alors le type de *var* est $\mathbb{P}$(T).

| Description | Ensembles | Types |
|---|---|---|
| Ensemble des entiers relatifs | $\mathbb{Z}$ | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des entiers | $\mathbb{N}$ | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des entiers non nuls | $\mathbb{N}_1$ | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des entiers concrets | *NAT* | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des entiers concrets non nul | $NAT_1$ | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des entiers relatifs concrets | *INT* | $\mathbb{P}(\mathbb{Z})$ |
| Ensemble des booléens | *BOOL* | $\mathbb{P}$(BOOL) |
| Ensemble des chaînes de caractères | *STRING* | $\mathbb{P}$(STRING) |

TAB. 2.6 – Type des ensembles prédéfinis

Dans le tableau 2.5, le typage des principales expressions constituant le langage B est décrit. Les types implémentables à travers une traduction [102] vers un langage compilable sont les suivants :

– les variables scalaires typées à l'aide de l'ensemble de type {$\mathbb{Z}$, *BOOL*, *STRING*},
– les tableaux à une dimension modélisés par une fonction totale de type $\mathbb{P}$(($\mathbb{Z}$ ∪ BOOL ∪ Ident) × ($\mathbb{Z}$ ∪ BOOL ∪ STRING ∪ Ident)).

Pour les tableaux les indices peuvent prendre les types $\mathbb{Z}$, BOOL ou Ident (type $T_{indice}$) tandis que les éléments du tableau peuvent prendre les types $\mathbb{Z}$, BOOL, STRING ou Ident

(type $T_{element}$) sachant que le type Ident représente les ensembles énumérés. Les tableaux à plusieurs dimensions seront donc modélisés par une fonction totale de type :
$\mathbb{P}((T_{indice_1} \times ... \times T_{indice_n}) \times T_{element})$.

### 2.3.7 Obligations de preuve

Le développement de systèmes par la méthode B donne lieu à l'écriture de machines abstraites. Les contraintes du cahier des charges sont définies par son invariant. Les obligations de preuves découlent de la préservation de l'invariant par la machine abstraite à travers la modification de son état par les opérations. A travers le raffinement, il faut préserver l'invariant pour éviter la disparition des contraintes du cahier des charges dans les implémentations, donc on obtient un nouvel ensemble d'obligations de preuve.

#### 2.3.7.1 Invariant

Il s'agit de prouver, à l'intérieur d'une même machine, que l'invariant est établi après l'initialisation et préservé par l'exécution des opérations. Je rappelle la structure d'une machine particulière sans paramètres avec une seule opération *OP* (fig 2.3).

| | |
|---|---|
| MACHINE | *M* |
| VARIABLES | *v* |
| INVARIANT | *I* |
| INITIALISATION | *U* |
| OPERATIONS | *OP* $\widehat{=}$ PRE *P* THEN *S* END |
| END | |

FIG. 2.3 – Une machine B

Pour prouver l'établissement de l'invariant *I* par l'initialisation *U* il suffit de prouver que le prédicat OP1 $\widehat{=}$ [U]I est satisfait. L'opération *OP* préserve l'invariant si le prédicat OP2 $\widehat{=}$ I $\wedge$ P $\Rightarrow$ [S]I est satisfait.

#### 2.3.7.2 Raffinement

Lorsqu'une machine *MachineA* (fig 2.4) est raffinée par une machine *MachineR* (fig 2.5), les propriétés de l'invariant de *MachineA* doivent être préservées. Elles font donc apparaître des obligations de preuve supplémentaires lors de l'utilisation du prouveur sur le raffinement *MachineR*.

Selon K. Lano [62] les obligations de preuves générées pour préserver l'invariant de *MachineA* durant le raffinement par *MachineR* sont les suivantes :

1. L'initialisation abstraite doit établir l'invariant concret si les contraintes (clause CONSTRAINTS) et propriétés (clause PROPERTIES) sont vraies :
   OP3 $\widehat{=}$ *condA* $\wedge$ *propA* $\wedge$ *propR* $\Rightarrow$ [*initR*] $\neg$ [*initA*] $\neg$ *invR*
2. L'opération abstraite doit établir l'invariant concret et l'égalité des résultats de l'opération concrète (*resR*) et de l'opération abstraite (*resA*) :
   OP4 $\widehat{=}$ *condA* $\wedge$ *propA* $\wedge$ *propR* $\wedge$ *invA* $\wedge$ *invR* $\wedge$ *preA* $\Rightarrow$ *preR* $\wedge$ [*subsR*] $\neg$ [*subsA*] $\neg$

| | |
|---|---|
| MACHINE | *MachineA (parMac)* |
| CONSTRAINTS | *condA* |
| CONSTANTS | *constA* |
| PROPERTIES | *propA* |
| VARIABLES | *varA* |
| INVARIANT | *invA* |
| INITIALISATION | *initA* |
| OPERATIONS | *resA*←*operation(paraOp)*≙PRE *preA* THEN *subsA* END |
| END | |

FIG. 2.4 – La machine abstraite *MachineA*

| | |
|---|---|
| MACHINE | *MachineR (parMac)* |
| REFINES | *MachineA* |
| CONSTANTS | *constR* |
| PROPERTIES | *propR* |
| VARIABLES | *varR* |
| INVARIANT | *invR* |
| INITIALISATION | *initR* |
| OPERATIONS | *resR*←*operation(paraOp)*≙PRE *preR* THEN *subsR* END |
| END | |

FIG. 2.5 – Le raffinement de *MachineA*

(*invR* ∧ *resA*=*resR*)
Ceci est à vérifier pour toutes les opérations.

Dans cette dernière obligation de preuve, on voit clairement que la pré-condition du raffinement doit être établie suivant la pré-condition de l'abstraction. Ceci illustre le concept d'affaiblissement de la pré-condition.

#### 2.3.7.3 Aparté sur la nécessité de la double négation

Dans le paragraphe précédent on remarque l'apparition de la double négation pour les deux dernières obligations de preuve. Nous allons ici donner l'intuition de la signification de cette double négation dans les obligations de preuve liées au raffinement. Une première interprétation de la nécessité de cette double négation est : un raffinement ne doit pas (première négation) être en désaccord (deuxième négation) avec son abstraction. Dans cette interprétation se trouve la double négation. Pour aller plus loin dans l'intuition, nous allons prendre un exemple de raffinement [79] et nous considérerons les chemins parcourus pour arriver au résultat :

$x := x + 2$ [] $x := x - 2$

L'application de cette substitution signifie qu'il est possible soit d'ajouter ou de retrancher 2 à $x$ (ceci est une substitution non déterministe du type CHOICE $x := x + 2$ OR $x := x - 2$

END). Un raffinement de cette spécification pourrait être :

$y := y + 1$ avec un invariant de collage $I \mathrel{\widehat{=}} y = 2 * x$

En considérant cette spécification, il faut démontrer d'après OP4 (en ne considérant que la partie sur les corps d'opération et en notant I l'invariant du raffinement) :

$[y := y + 1]\neg[x := x + 2\ []x := x - 2]\neg I$

En développant la substitution de choix :

$[y := y + 1]\neg([x := x + 2]\neg I \wedge [x := x - 2]\neg I)$

Cette expression s'interprète par : les deux chemins possibles doivent être valides. En continuant le développement, on obtient :

$[y := y + 1]\neg(\neg[x := x + 2]I \wedge \neg[x := x - 2]I)$

$[y := y + 1](\neg\neg[x := x + 2]\mathrm{I} \vee \neg\neg[x := x - 2]I)$

$([y := y + 1][x := x + 2]I) \vee ([y := y + 1][x := x - 2]I)$

Ce qui peut s'interpréter par : la substitution de raffinement doit être cohérente avec l'une ou l'autre alternative de l'abstraction.

### 2.3.8 Les mécanismes de composition

La modélisation des systèmes en B utilise la notion de modularité pour isoler les contraintes afin d'en réduire la complexité. Dans le but de créer de grosses spécifications tout en utilisant la modularité, la méthode B utilise des mécanismes de composition par les clauses suivantes :

- SEES
- USES
- INCLUDES
- EXTENDS
- PROMOTES
- IMPORTS

Dans les paragraphes suivants, les règles de visibilité [93] entre un composant *C1* et un composant *C2* définissent pour chaque constituant (variable, ensemble, constante et opérations) de *C2*, les modes d'accès applicables dans les clauses de composition de *C1*. Pour des données, on distingue l'accès en lecture seule, en lecture écriture ou en écriture seule. Pour des opérations, on distingue l'accès aux opérations de consultation (les opérations dont la spécification ne modifie pas les variables de la machine) et aux opérations de modification. Les règles de visibilité montrent quatre types de visibilité (Tableau 2.7).

- Les constituants peuvent être visibles comme un ensemble ou une variable dans un invariant (l'invariant ne modifie pas les valeurs des variables) qui est annoté **V**.
- Le cas des variables ou opérations est plus complexe on dira qu'une variable est visible et non modifiable ce qui est annoté **N**.
- Elles peuvent être visibles et modifiables ce qui est annoté **M**.

Les opérations peuvent être appelées lorsqu'elles sont visibles et non modifiables dans le cas où elles ne modifient pas la valeur des variables de la machine ou lorsqu'elles sont visibles et modifiables.

#### 2.3.8.1 SEES

La clause SEES permet à un composant de voir une instance de machine. Elle permet d'accéder aux constituants de la machine vue sans possibilité de les modifier. La clause SEES permet de voir l'instance la plus proche dans le graphe de dépendance du projet B. Dans

| Mode de visibilité | Description |
|---|---|
| | constituant non visible |
| V | constituant visible |
| N | constituant visible,<br>si le constituant est une variable utilisée dans une substitution,<br>la variable est modifiable,<br>si le constituant est une opération appelée dans une substitution,<br>l'opération peut modifier les variables de sa machine abstraite |
| M | constituant visible,<br>si le constituant est une variable utilisée dans une substitution,<br>la variable n'est pas modifiable,<br>si le constituant est une opération appelée dans une substitution,<br>c'est une opération qui ne modifie pas les variables de sa machine abstraite |

TAB. 2.7 – Type de visibilité

l'exemple suivant (fig 2.6) la machine vue par *MA* sera celle importée par *MD* et non pas toutes ou celles importées par *ME*. La règle consiste à dire que la machine vue est l'instance importée par le plus proche parent de la machine qui voit. On peut aussi renommer les machines vues.

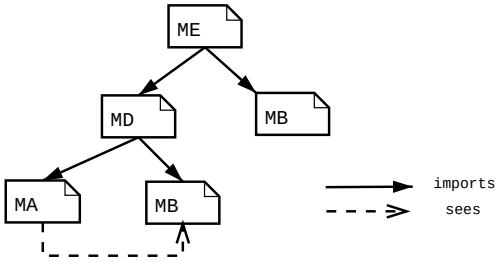


FIG. 2.6 – Graphe de dépendance d'un projet B

Les règles de visibilité de la clause SEES sont définies dans le tableau 2.8 où la machine *MA* voit la machine *MB*. Il existe une restriction sur la clause SEES : elle n'est pas transitive c'est-à-dire qu'une machine *MA* voit une machine *MB* qui voit une machine *MC* mais *MA* ne voit pas la machine *MC*. En outre une machine vue doit être importée (instanciation par la clause IMPORTS) dans le projet B.

#### 2.3.8.2 INCLUDES, EXTENDS et PROMOTES

Le lien INCLUDES entre un composant *MA* (machine ou raffinement) et un composant *MB* permet d'inclure une instance de *MB* dans *MA* afin de construire un composant plus volumineux tout en gardant les avantages des aspects de modularité. Aussi la clause EXTENDS possède les mêmes propriétés que la clause INCLUDES et ajoute la promotion de toutes les opérations des instances de machines étendues (incluses). Dans le cas où l'on ne veut pas promouvoir toutes les opérations d'une machine dans un composant abstrait (machine

| Constituants de MB | Clauses de MA | | | |
|---|---|---|---|---|
| | CONSTRAINTS | PROPERTIES | INVARIANT | OPERATIONS |
| Paramètres | | | | |
| Ensembles | V | V | V | V |
| Constantes | V | V | V | V |
| Variables | | | | N |
| Opérations | | | | N |

TAB. 2.8 – Visibilité des entités d'une machine vue

ou raffinement) mais certaines il faut donc utiliser la clause INCLUDES et promouvoir les opérations considérées avec la clause PROMOTES. Les clauses INCLUDES et EXTENDS sont transitives c'est-à-dire que dans le cas où une machine *MA* qui inclut ou étend une machine *MB* qui fait la même chose pour *MC* alors *MA* peut utiliser les constituants de *MC* comme ceux de *MB*. Les règles de visibilité des clauses INCLUDES et EXTENDS sont décrites dans le tableau suivant (Tab 2.9) sachant que *MA* inclut *MB* ou *MA* étend *MB*.

| Constituants de MB | Clauses de MA | | | |
|---|---|---|---|---|
| | INCLUDES | PROPERTIES | INVARIANT | OPERATIONS |
| Paramètres | | | | |
| Ensembles | | V | V | V |
| Constantes | | V | V | V |
| Variables | | | V | N |
| Opérations | | | | M |

TAB. 2.9 – Visibilité des entités d'une machine incluse ou étendue

#### 2.3.8.3 USES

Lorsqu'un composant inclut plusieurs instances de machine (clause INCLUDES), les machines incluses peuvent partager des données de l'une d'entre elles par un lien USES sur la machine propriétaire des données. L'avantage du lien USES est qu'il permet d'utiliser les paramètres formels de la machine utilisée (Tab 2.10). La clause USES n'est pas transitive.

| Constituants de MB | Clauses de MA | | | |
|---|---|---|---|---|
| | CONSTRAINTS | PROPERTIES | INVARIANT | OPERATIONS |
| Paramètres | | | V | V |
| Ensembles | | V | V | V |
| Constantes | | V | V | V |
| Variables | | | V | N |
| Opérations | | | | |

TAB. 2.10 – Visibilité d'une machine utilisée

#### 2.3.8.4 IMPORTS et PROMOTES

Le lien IMPORTS entre une implantation *MA* et une machine *MB* permet d'une part d'instancier la machine *MB* dans le projet et d'autre part de disposer entièrement des services de *MB* qui peuvent être promus par la clause PROMOTES comme pour l'inclusion. L'utilisation des services de *MB* est réalisée à travers l'appel des opérations de *MB* sachant que l'on se place dans le cadre du B logiciel et que les opérations sont exécutées par leur appel (elles ne s'exécutent pas automatiquement comme les événements lorsque leur garde est vraie). Par conséquent la machine qui importe contrôle les moyens d'écriture des données modifiables de la machine importée. L'importation permet de structurer un projet B en couches, en effet l'implantation d'un module s'implémente par importation d'autres modules offrant des services de plus bas niveau. D'autre part le composant qui n'est pas importé joue un rôle particulier au sein d'un projet B : il est le point d'entrée de son exécution. Le tableau suivant (Tab 2.11) décrit la visibilité des constituants d'une machine importée *MB* à l'intérieur des clauses d'une machine *MA*.

| Constituants de MB | Clauses de MA | | | |
|---|---|---|---|---|
| | PROPERTIES | VALUES | INVARIANT | OPERATIONS |
| Paramètres | | | | |
| Ensembles et Constantes Concrètes | V | V | V | V |
| Constantes Abstraites | V | | V | |
| Variables Abstraites | | | V | |
| Variables Concrètes | | | V | N |
| Opérations | | | | M |

TAB. 2.11 – Visibilité des entités d'une machine importée

## 2.4 Conclusion

La méthode B est basée sur une sémantique et une syntaxe mathématiques très précises. Elle permet le développement d'applications prouvées. Le cycle de développement B consiste à définir dans l'invariant les propriétés (contraintes mathématiques) que le système doit respecter, et à écrire les opérations qui feront évoluer l'état du système. Les différents modèles construits dans cette optique font apparaître graduellement par raffinement un modèle concret traduisible dans un langage de programmation traditionnel tout en maintenant le respect des propriétés des modèles abstraits. Cependant, elle souffre d'un manque de compréhension de la part des non initiés et donc d'une difficulté d'apprentissage et d'utilisation. Les mécanismes de composition des machines ont été choisis pour la préservation des preuves mais ils sont inhabituels pour la structuration des informations.

# Deuxième partie

# Extraction de vues UML

# Chapitre 3

# Contexte et motivations

Dans les paragraphes précédents nous avons choisi la méthode formelle adéquate à ce travail de recherche. Nous rappelons que les méthodes formelles et semi-formelles ont chacune leurs avantages sans qu'aucune ne permettent de répondre complètement aux besoins du développement logiciel.

## 3.1 Utilisation conjointe des méthodes formelles et semi-formelles

Les notations semi-formelles sont intuitives et synthétiques mais ambiguës au contraire des méthodes formelles qui sont précises et rigoureuses mais complexes. Ces deux approches sont complémentaires et aucune des deux n'est à abandonner car il est intéressant de combiner les avantages respectifs de l'une et de l'autre. Se pose maintenant la question du choix de la phase de développement qui est la plus appropriée à leur intégration. Dans toutes les méthodes semi-formelles classiques on retrouve les phases suivantes :

- analyse des besoins,
- spécification,
- conception et développement.

La phase de développement, où l'intégration devrait être la plus bénéfique, est celle de la spécification où la précision, l'absence d'ambiguïté et la preuve du fonctionnement sont particulièrement importantes. D'autre part la lisibilité des diagrammes des méthodes semi-formelles apporte une vue synthétique éclairant les spécifications formelles que les notations mathématiques obscurcissent. Concernant l'intégration des méthodes formelles et semi-formelles, il existe trois types de démarches :

1. La première démarche vise à proposer une nouvelle méthode qui unifie les deux approches. Il existe deux moyens de l'obtenir :
   - La première forme permet de définir une nouvelle méthode cohérente qui étend les concepts des notations semi-formelles existantes en y ajoutant des bases formelles solides. Deux méthodes répondent à ce schéma, la méthode Fusion [13] dans laquelle les opérations sont décrites notamment en utilisant la structure *pré* et *post-condition* existante dans de nombreuses méthodes formelles et l'approche d'Attiogbé Poizat et Salaün [38] qui permet d'ajouter des concepts formels provenant de spécifications B, Z ou algébriques dans des diagrammes d'états UML.

- La deuxième forme consiste en un schéma complètement opposé en incorporant les notions orientées-objets dans les méthodes formelles telles que VDM ou Z qui deviennent VDM++ [24] Z++ [59] et ObjectZ [21]. De cette manière on conserve les avantages au niveau précision des méthodes formelles tout en structurant la spécification vers un langage plus proche des langages impératifs traditionnels à objets.

2. La deuxième démarche est l'obtention d'une spécification formelle à partir d'une notation semi-formelle, cette démarche est appelée dérivation. Beaucoup de travaux s'inscrivent dans ce schéma parmi lesquels nous pouvons citer [22, 23, 42] qui utilisent la méthode formelle Z. D'autre part d'autres travaux utilisent la méthode B comme [76, 61, 77, 64, 73]. Ces deux groupes de travaux utilisent sauf exception les mêmes types de diagrammes semi-formels. Premièrement ils dérivent les données statiques du diagramme de classes. Ensuite ils utilisent la spécification du comportement du système défini dans les diagrammes d'états-transitions pour dériver les opérations qui modifieront l'état du système.
3. La troisième démarche consiste à extraire les vues graphiques semi-formelles depuis la spécification formelle. Au contraire de la démarche précédente, à notre connaissance, peu de travaux concernent cette démarche. Deux travaux [18] et [70] traitent ce type de démarche. Le premier présente un outil de traduction sous forme de graphes d'une spécification écrite en VDM [58]. Le second définit une méthode de représentation d'expressions relationnelles par des graphes et propose un algorithme qui calcule la formule relationnelle sans quantificateur interprétant le graphe. Ces travaux sont spécifiques aux modèles relationnels pour les bases de donnés. Plus récemment une approche a généralisé ces travaux au génie logiciel comme [32].

Remarque : il existe d'autres travaux autour de la construction simultanée de vues UML et de modèles B dans l'approche proposée par Okalas Ossami, Jacquot et Souquières [30].

## 3.2 Motivations

L'ajout de fondements formels aux méthodes semi-formelles ou de notations graphiques aux méthodes formelles est très complexe à obtenir. En effet les philosophies employées dans ces deux méthodes sont très différentes. De plus le travail à mettre en œuvre dans ce genre de démarche est très important sans être sûr d'un résultat probant. En outre proposer une nouvelle méthode pose des problèmes financiers et humains : elle nécessite une reconstruction des applications existantes et implique une formation spécifique des utilisateurs dont les habitudes de travail peuvent radicalement changer sans oublier le coût de fabrication des outils associés.

La deuxième démarche sur laquelle de nombreux travaux [40, 60, 84, 74, 63, 36, 73] ont été menés ces dernières années promeut la dérivation de spécifications B à partir de différents diagrammes OMT et UML. Les avantages d'une telle approche sont évidents : la modélisation est réalisée en dehors du cadre de la méthode formelle et permet aux concepteurs de l'ignorer. Les inconvénients sont de deux types :

- La notation B n'est pas orientée-objet mais modulaire par conséquent la spécification abstraite engendrée à partir d'une modélisation à objets est éloignée de celle qui aurait été écrite directement en utilisant une méthode formelle.
- La spécification découlant de la dérivation est incomplète car les méthodes semi-formelles ne possèdent pas les notions de substitutions. Il faudra donc modifier une spécification

générée automatiquement puis poursuivre par l'écriture des raffinements et prouver chacune des machines.

Ceci conduit à penser qu'une partie des concepteurs choisira d'écrire directement ses spécifications en B. Il devient alors intéressant de pouvoir extraire de celles-ci les éléments permettant de mieux les comprendre, les présenter et les documenter. Ces éléments seront exprimés dans une notation de modélisation graphique de plus en plus utilisée dans le domaine de la conception de logiciels c'est-à-dire UML qui, au delà des vues graphiques, dispose d'un langage de contraintes appelé OCL [103] (Object Constraints Language) qui permet de spécifier des invariants (de classe) et des pré et post-conditions de méthodes. OCL permet de préciser les vues UML graphiques. En outre il faut noter que si plusieurs démarches existent dans le domaine de la dérivation d'UML vers B, aucune n'existe dans l'autre sens. Dans le cadre d'une démarche itérative de développement elle serait indispensable et permettrait la traçabilité entre la modélisation et la spécification.

## 3.3 Contenu de la deuxième partie

Nous commencerons par une brève présentation de la notation UML et de son langage de description des contraintes : OCL. Après un état de l'art dédié principalement à la dérivation de vues UML en spécification B, nous présenterons notre première contribution autour de l'extraction de vues UML depuis une spécification B. Cette contribution est décrite dans quatre chapitres. Deux chapitres traitent de la partie statique des spécifications UML. Le premier chapitre traitera de l'extraction de diagrammes de classes suivi du deuxième de traduction des propriétés invariantes B en contraintes OCL afin de préciser le diagramme de classes. Les deux chapitres suivant définiront l'extraction des données dynamiques du système par les diagrammes d'états-transitions, de séquence et d'activités. Enfin une conclusion achèvera le document sur l'intégration des méthodes formelles et semi-formelles par extraction.

# Chapitre 4

# La notation UML

## 4.1 Introduction

Les méthodes semi-formelles ont été conçues pour modéliser les systèmes d'information dans le but d'en obtenir une description moins ambiguë et plus synthétique qu'en langage naturel. Nous pouvons citer quelques méthodes comme : E/R [11], NIAM [47], OMT [82] et UML [35]. Deux modèles au minimum sont toujours proposés par ces méthodes :

- L'un décrit l'aspect statique, c'est-à-dire la structure des données et leurs relations.
- L'autre décrit les traitements qui ont lieu sur les données : il s'agit de l'aspect dynamique.

Certaines méthodes comportent d'autres modèles comme dans la méthode OMT où le modèle fonctionnel permet de décrire la transformation des valeurs des données dans le système.
Dans ce chapitre, le langage UML est présenté. Cette notation est née de la fusion de plusieurs méthodes objets telles que entre autres OMT [82], Booch [10] et OOSE [56]. Il a été normalisé pour la première fois en 1997 [88] par l'OMG (Object Management Group[1]). Le chapitre commence par une présentation générale du langage puis continue par la présentation des différents diagrammes et se termine par le langage OCL.

## 4.2 Généralités

La notation UML permet la construction de modèles. La modélisation est réalisée à l'aide de diagrammes. Les diagrammes permettent une visualisation graphique du système. UML dispose de neuf diagrammes sachant que chacun d'entre eux possède un rôle précis dans la modélisation du système :

- **Diagramme de classes** : Il représente les données du modèle (partie statique). Il fait abstraction des parties dynamiques et temporelles. Il peut être instancié à travers un diagramme d'objets. Il est composé de classes et associations.
- **Diagramme d'objets** : Il représente un diagramme de classes instanciées. Dans ce diagramme les classes deviennent plusieurs objets et les associations plusieurs liens entre objets. Il correspond à un diagramme de collaboration sans les interactions matérialisées par des messages.
- **Diagramme de cas d'utilisation** : Il permet de matérialiser et structurer les besoins des utilisateurs du système. Il identifie les acteurs (utilisateurs du système) et leurs

[1] http ://www.omg.com

interactions avec le système. L'acteur consulte ou modifie l'état du système. Le système réagit aux demandes de l'acteur par les différents cas d'utilisation pour lesquels il est prévu.

- **Diagramme de composants** : Il permet de décrire les composants et leurs dépendances dans l'environnement de réalisation. Les diagrammes de composants sont des vues statiques de l'implémentation des systèmes qui montrent le choix de réalisation.
- **Diagramme de déploiement** : Il montre la disposition physique et matérielle qui compose le système et la répartition des composants sur le matériel. Les ressources matérielles sont représentées sous forme de noeuds, lesquels sont connectés entre eux par un support de communication et leurs caractéristiques peuvent être précisées. Ces diagrammes représentent le déploiement des composants sur le dispositif matériel.
- **Diagramme de collaboration** : Il représente les interactions entre les acteurs et les objets (instances de classes). Il permet de spécifier de manière très précise l'ordre et les conditions d'envoi des messages de communication entre les objets.
- **Diagramme de séquence** : Il représente les communications entre les objets selon un point de vue temporel. Dans ce diagramme, on met l'accent sur la chronologie des envois de messages. L'ordre d'envoi des messages est défini par son emplacement sur la ligne de vie des objets qui correspond au temps qui s'écoule.
- **Diagramme d'états-transitions** : Il représente les modifications de l'état d'un objet selon son propre comportement et/ou le comportement des autres objets du système. Comme son nom l'indique le diagramme d'états-transitions est composé d'états et d'arcs orientés appelés transitions qui déterminent les changements d'états.
- **Diagramme d'activités** : Le diagramme d'activité est une variante du diagramme d'états-transitions. En effet au contraire du diagramme d'états-transitions qui définit le comportement d'un objet ou composant, le diagramme d'activité définit les actions exécutées à l'intérieur d'une méthode.

Dans la suite de ce chapitre nous allons présenter en détails les différents diagrammes nécessaires à l'extraction des vues UML depuis une spécification B.

## 4.3 Diagramme de classes

Le diagramme de classes montre une vue statique du système. Il est le constituant le plus utilisé de la vue statique. Il fait abstraction des aspects dynamiques et temporels. Cependant il montre quelques aspects dynamiques du système comme les méthodes mais il n'en décrit pas le comportement. Il contient entre autres les éléments suivants :

- les classes composées d'attributs et de méthodes,
- les généralisations,
- les agrégations, compositions et associations.

### 4.3.1 Classes, attributs et méthodes

Les classes représentent un groupe d'objets qui ont les mêmes données et les mêmes comportements. Une classe est une description abstraite de ces objets. Chaque classe est représentée sous la forme d'un rectangle divisé en trois compartiments. Le compartiment supérieur comprend le nom de la classe. Le compartiment médian contient les attributs. Le dernier item représente le comportement d'un groupe d'objets, il est défini à l'aide de méthodes.

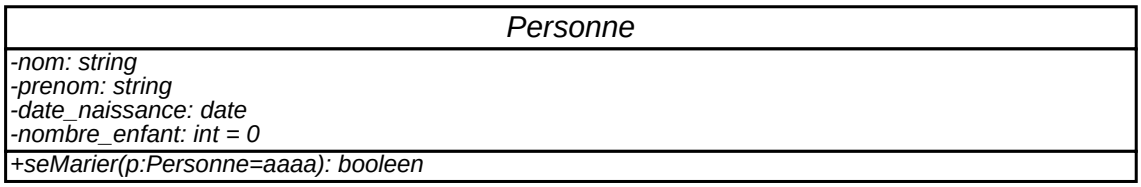


FIG. 4.1 – Une classe pourvue d'attributs et de méthodes

La visibilité des attributs et méthodes est définie par les trois caractères suivants : '-', '+' et '#'. Le caractère '-' (*private*) signifie que l'item est privé c'est-à-dire qu'il est visible uniquement à l'intérieur de la classe. Le caractère '+' (*public*) signifie au contraire qu'il est visible à l'extérieur autant qu'à l'intérieur. Enfin '#' (*protected*) signifie que les classes filles hériteront de cet item à travers une généralisation.

Les attributs sont des propriétés nommées d'une classe. Ils décrivent les données des objets. On obtient par concaténation de leur valeurs l'état de l'objet instancié de la classe. Ils sont décrits par une chaîne de caractères qui comprend les notions suivantes :

[visibilité]nom[ :type][=valeur_initiale]

Un attribut est obligatoirement nommé par contre sa visibilité, son typage et sa valeur initiale sont facultatifs.

Les méthodes définissent le comportement des objets cependant elles sont seulement citées et ne participent pas à la description du comportement des objets à travers le diagramme de classe. Les méthodes sont spécifiées par le prototype suivant :

[visibilité]nom([paramètre[ :type][=valeur_par_défaut]]*)[ :type]

Tout comme pour les attributs le typage de la valeur de retour, le typage et la valeur par défaut des différents paramètres ne sont pas obligatoires. En outre une méthode peut ne prendre aucun paramètre ni retourner de valeur.

### 4.3.2 Généralisation

La généralisation est un processus qui permet de définir des données et un comportement commun entre plusieurs classes. Elle permet de définir une classe de base appelée classe *mère*. Par la suite seules les différences entre les classes *filles* apparaîtront dans les classes *filles*. Les variables et méthodes dont la visibilité est déclarée *protected* seront visibles uniquement dans la classe fille. Par conséquent le typage des attributs, paramètres et valeurs retournées par les méthodes, les prototypes de méthodes, les contraintes (valeurs par défaut et valeurs initiales) et les associations relatives à la classe *mère* sont maintenues dans la classe *fille*. Une classe *fille* peut intervenir dans plusieurs généralisations ; on parle alors de généralisation multiple. La généralisation est représentée par un triangle sur son extrémité du côté de la classe *mère* et un arc qui est relié à la classe *fille* (fig 4.2).

### 4.3.3 Association, qualification, multiplicité, agrégation et composition

#### 4.3.3.1 Association

Une association représente une relation entre deux ou plusieurs classes d'objets. Si l'association relie deux classes, on parle d'association binaire, elle est représentée par un trait

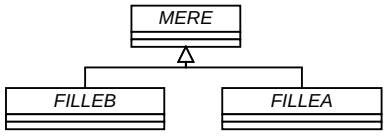


FIG. 4.2 – Généralisation

entre les deux classes (fig 4.3). Si l'association relie trois classes ou plus on parle d'association n-aire, elle est représentée par un losange relié à chaque classe par un trait. L'intersection entre l'extrémité d'une association et une classe peut porter un nom que l'on appelle rôle. De plus une association peut comporter des attributs que l'on appelle attributs de lien. Dans ce cas on crée une classe qui ne possède pas de nom que l'on relie à l'association par une ligne en pointillés.

#### 4.3.3.2 Multiplicité

La multiplicité concerne le nombre d'objets mis en jeu pour un lien (fig 4.5) d'une association. Elle répond à la question : combien d'objets de la classe considérée peuvent être mis en relation avec un objet de l'autre extrémité de l'association ? Elle est définie par un entier ou un intervalle sous les formes suivantes :

- N où N $\in [1,+\infty[$
- N..M où N $\in [1,+\infty[$ , M $\in [1,+\infty[$ et M>N
- N..* où N $\in [1,+\infty[$ sachant que les notation 0..* et * sont équivalentes

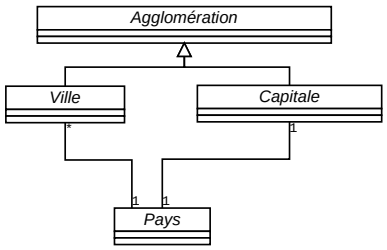


FIG. 4.3 – Généralisation, association et multiplicité

### 4.3.4 Qualification des associations

La qualification d'une association, aussi dénommée restriction d'une association, consiste à sélectionner un sous-ensemble d'objets parmi l'ensemble des objets qui participent à une association. La restriction est réalisée au moyen d'un tuple d'attributs particuliers, appelé qualificatif ou clé, qui est utilisé conjointement avec un objet de la classe source.

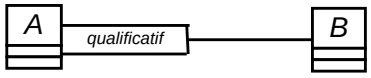


FIG. 4.4 – Qualification d'association

L'instanciation d'une association qualifiée définit le nom des objets source et destination, et la valeur du qualificatif. Ainsi, chaque instance de la classe $A$, accompagnée de la valeur du

qualificatif, identifie un sous-ensemble des instances de *B* qui participent à l'association. La qualification partitionne l'ensemble d'arrivée et réduit ainsi la multiplicité de l'association.

#### 4.3.4.1 Agrégation

Une agrégation est un cas particulier d'association où une classe est composée de l'autre. Elle est représentée par un losange du côté de la classe composée et un trait qui le relie à la classe composante. Elle représente une association asymétrique dans laquelle l'une des classes est composante de l'autre sans que l'autre ne soit tributaire de son existence. Au contraire, la composition est un cas particulier d'agrégation qui nécessite l'existence de l'autre. Elle est représentée par la même symbolisation que l'agrégation à l'exception du losange qui est plein. En terme de multiplicité, l'extrémité de l'agrégation ou de la composition du côté de la classe composante est ouverte dans les deux cas tandis que de l'autre, du côté composé, elle est généralement chiffrée.

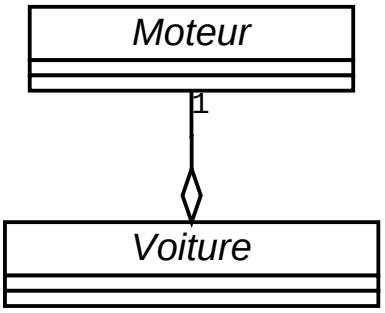


FIG. 4.5 – Une agrégation

## 4.4 Diagramme d'états-transitions

Les diagrammes d'états-transitions décrivent une vue dynamique du système. Ils sont représentés par des automates à états finis composés de transitions, d'événements et d'activités. La sémantique et la notation sont celles des statecharts de Harel [50].

### 4.4.1 Etats

Un état est représenté par un rectangle aux coins arrondis à l'exception des états initiaux et finaux. L'état initial est représenté par un disque noir et l'état final par un disque plein entouré par un cercle. Excepté les états initiaux et finaux, chaque état peut posséder les activités suivantes :

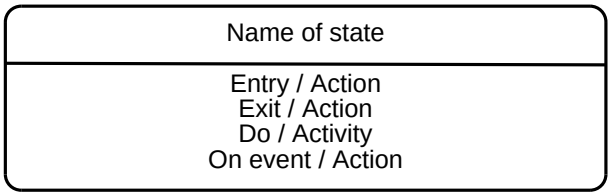


FIG. 4.6 – Notation d'un état en UML

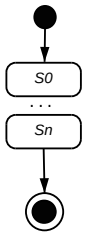


FIG. 4.7 – Notation d'un diagramme d'états-transitions

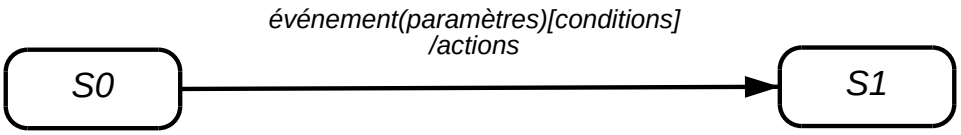


FIG. 4.8 – Notation d'une transition complexe

- **entry / action** : action exécutée à l'entrée de l'état
- **exit / action** : action exécutée à la sortie de l'état
- **on event / action** : action exécutée à chaque fois que l'événement cité survient
- **do / activity** : action récurrente ou significative, exécutée dans l'état

D'autre part un état peut contenir un diagramme d'états-transitions interne dont l'état final doit être atteint pour exécuter toute autre transition externe à l'état. Dans ce cas l'état englobant le sous-diagramme est appelé super-état. La notion de super-état peut aussi être utilisée lorsqu'il contient le diagramme complet d'une variable ou d'un système.

### 4.4.2 Transitions

Une transition est représentée par une flèche dont l'origine est reliée à l'état source et dont la destination est connectée à l'état cible. Elle est décorée par le nom de l'événement déclencheur suivi éventuellement de paramètres, de conditions de déclenchement (garde) et d'une liste d'actions à effectuer. Les paramètres sont décrits entre parenthèses. La condition de déclenchement est exprimée entre crochets après le nom de l'événement et la liste d'actions est séparée des trois précédents éléments par le caractère '/'.

### 4.4.3 Concurrence

La concurrence peut être exprimée sous deux formes dans les diagrammes d'états-transitions :

1. Des variables du système peuvent être en concurrence dans ce cas une condition de transition d'une des deux variables est exprimée en fonction de l'état du diagramme de l'autre comme pour les variables *X* et *Y* de la figure 4.9. En effet dans le diagramme d'états-transitions de *Y* le passage de l'état *y1* à l'état *y2* est conditionné par *[IN X = x1]* en référence à l'état du diagramme d'états-transitions de *X*. Pour que la transition de *y1* à *y2* puisse s'exécuter, il faut que l'état courant du diagramme de *X* soit *x1*.
2. Deux sous-diagrammes peuvent être en concurrence et dans ce cas les deux sous-diagrammes seront situés à l'intérieur d'un état du diagramme d'états-transition du système (fig 4.10).

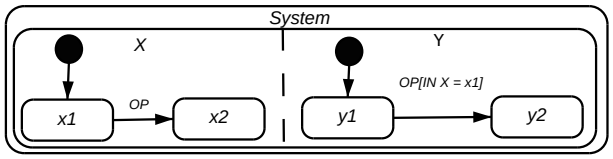


FIG. 4.9 – Diagrammes d'états-transitions concurrents

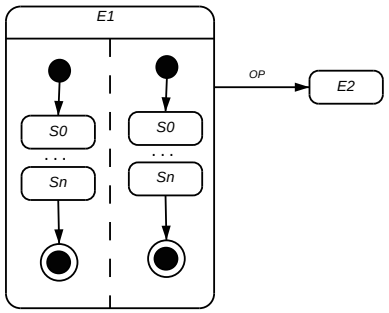


FIG. 4.10 – Diagrammes concurrents au sein d'un état

## 4.5 Diagramme de séquence

Les diagrammes de séquences permettent de modéliser les aspects dynamiques des systèmes. Ils montrent les interactions entre les objets. La représentation se concentre sur la séquence d'interaction d'un point de vue temporel. Elle se traduit par l'envoi de messages entre les différents objets du système.

### 4.5.1 Les objets

Les acteurs d'un diagramme de séquence sont les objets du système définis dans le diagramme d'objet instancié à partir des classes du diagramme de classes. Dans ce diagramme, les événements extérieurs ne sont pas pris en compte. Les seuls phénomènes décrits sont les interactions entre les différents objets du système. Un objet est matérialisé par un rectangle nommé par son nom. On y ajoute une barre verticale qui correspond à sa ligne de vie et qui représente un axe temporel permettant de dater les communications.

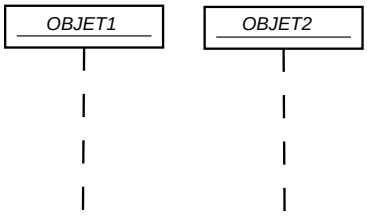


FIG. 4.11 – Les acteurs d'un diagramme de séquence

### 4.5.2 Messages

Les messages permettent la communication entre les différents objets du système. Ils correspondent souvent à l'appel d'une méthode d'un objet par celle d'un autre. Ils peuvent aussi signifier l'envoi de données, la création ou la destruction d'un objet ou d'autres opérations. Un objet recevant un message peut y répondre sans que cela soit systématique.

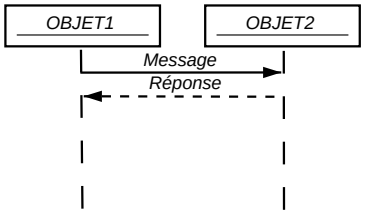


FIG. 4.12 – Les messages d'un diagramme de séquence

### 4.5.3 Temps d'exécution

Lorsqu'un objet *OBJET1* envoie un message à un autre objet *OBJET2*, ce message peut être un appel d'exécution d'une méthode. Dans ce cas, le temps qui s'écoule entre l'arrivée du message et le renvoi éventuel de la réponse est un temps d'exécution que l'on représente par un rectangle sur la ligne de vie de l'objet concerné. D'autre part la demande d'exécution de la méthode provient d'une méthode du premier objet qui s'exécute selon une durée qui englobe la durée d'exécution de la méthode appelée.

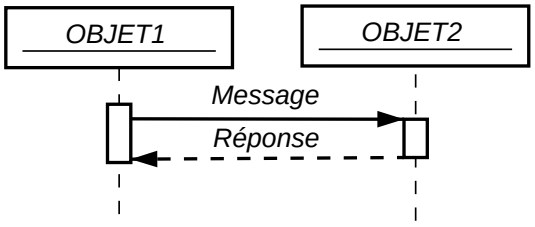


FIG. 4.13 – Diagramme de séquence avec temps d'exécution des méthodes

### 4.5.4 Conditions

Lors de l'utilisation d'expressions conditionnelles la ligne de vie peut être dédoublée (fig 4.14) pour permettre d'exécuter l'une ou l'autre méthode à la même date. Les messages qui déterminent l'exécution conditionnelle des deux méthodes ont la même origine. La condition est écrite entre crochets devant l'appel de la méthode.

### 4.5.5 Pseudo-code

On peut aussi ajouter des lignes de pseudo-code en marge de la ligne de vie de l'objet comme dans l'exemple suivant (fig 4.15). Ici on utilise du pseudo-code pour modéliser une boucle.

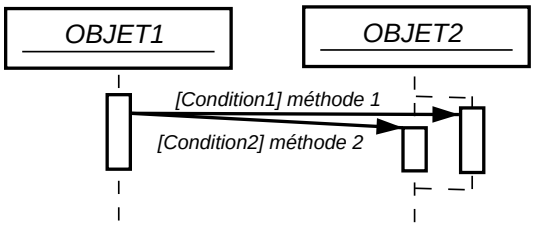


FIG. 4.14 – Utilisation de conditions dans un diagramme de séquence

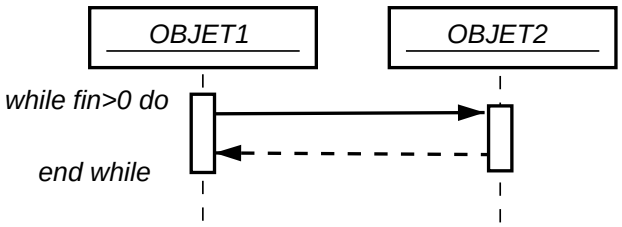


FIG. 4.15 – Utilisation de pseudo-codes dans un diagramme de séquence

### 4.5.6 Contraintes temporelles

Les contraintes temporelles sont décrites à l'aide de repères temporels sur les lignes de vie des objets et par une condition entre crochets en utilisant les repères en marge du diagramme (fig 4.16).

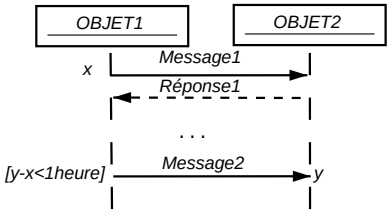


FIG. 4.16 – Utilisation de contraintes temporelles dans un diagramme de séquence

## 4.6 Diagramme d'activités

Un diagramme d'activité est une variante des diagrammes d'états-transitions. Dans un diagramme d'états-transitions, les états et les transitions sont mis en avant alors que dans un diagramme d'activités, ce sont les activités qui sont mises en avant. Les deux types de diagrammes permettent ainsi d'avoir deux vues différentes sur les automates du système. Un diagramme d'activités visualise un graphique d'activités qui modélise le comportement d'une méthode. Les diagrammes d'activités utilisent beaucoup d'éléments des diagrammes d'états-transitions en tant que variante de ce dernier. Un diagramme d'activités représente l'état de l'exécution d'un mécanisme, sous la forme d'un déroulement d'étapes regroupées séquentiellement dans des branches parallèles de flots de contrôle. Le début et la fin d'un mécanisme sont définis respectivement par un état initial et un état final facultatif. Les acti-

vités d'états se notent de la même manière que dans les diagrammes d'états-transitions selon qu'elles sont exécutées en entrée (entry) durant l'exécution de l'état (do) ou en sortie (exit) (fig 4.19).

### 4.6.1 Les objets

Les acteurs du diagramme sont les objets du système. Ils sont représentés par des travées nommées par le nom de l'objet. Chaque passage d'une travée à l'autre par une transition est considéré comme un appel de méthode d'un objet par une autre de l'objet relatif à la travée (fig 4.17).

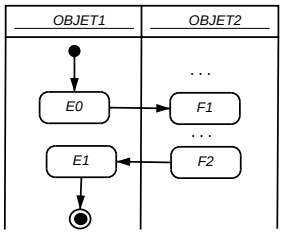


FIG. 4.17 – Les acteurs d'un diagramme d'activité

### 4.6.2 Exécutions concurrentes

Les exécutions concurrentes dans un diagramme d'activité ne se notent pas de la même manière que dans les diagrammes d'états-transitions. On utilise une barre horizontale pour définir le fait que l'on fractionne le flot d'exécution en plusieurs (fig 4.18).

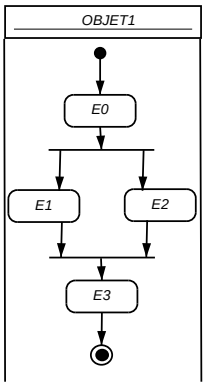


FIG. 4.18 – Exécutions concurrentes de branches d'un diagramme d'activité

### 4.6.3 Points de choix et conditions

Un point de choix est une condition vraie ou fausse qui détermine l'orientation du flot d'exécution entre plusieurs branches du diagramme. Ils sont représentés par un losange avec une condition, entre crochets, appliquée à la transition comme *[i<10]* dans l'exemple suivant (fig 4.19).

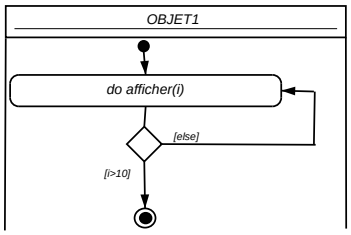


FIG. 4.19 – Les points de choix dans les diagrammes d'activité

## 4.7 Le langage OCL

OCL (Object Constraint Language) est une contribution d'IBM à UML 1.1 [103]. Comme son nom l'indique, il permet d'exprimer les contraintes dans le contexte UML. Les contraintes d'un système sont des informations supplémentaires importantes qui n'ont pas pu être spécifiées à l'aide d'éléments de modélisation définis dans les diagrammes UML. Ce langage formel a été conçu pour qu'il soit compréhensible par des informaticiens, pour qu'il s'intègre dans la notation orientée objet d'UML et pour éviter des notations mathématiques. OCL permet ainsi de limiter les ambiguïtés tout en restant accessible. Il exprime :

- les invariants au sein d'une classe ou d'un type : contraintes qui doivent toujours être vérifiées pour s'assurer du bon fonctionnement des instances de la classe ou du type concerné ;
- les contraintes au sein d'une opération : contraintes qui doivent toujours être vérifiées pour s'assurer de la bonne exécution de l'opération concernée ;
- les pré et post-conditions d'opération : contraintes qui doivent être respectivement vérifiées avant et après l'exécution d'une opération ;
- les gardes : contraintes sur la modification de l'état d'un objet ;
- les expressions de navigation : contraintes pour représenter les chemins au sein de la structure de classes.

### 4.7.1 Le contexte

Chaque contrainte OCL est liée à un contexte définissant le type auquel la contrainte se rapporte. Chaque instance d'un type doit ensuite respecter les contraintes définies pour ce type. Le contexte d'une expression OCL est défini explicitement avec le mot clé **context** suivi du nom de l'élément de modélisation concerné et du stéréotype de la contrainte. Il est suivi d'un " :", puis de l'expression de la contrainte :

- le mot clé **inv** correspond au stéréotype *invariant* alors que les mots clés **pre** et **post** sont utilisés pour indiquer respectivement les stéréotypes *pré-condition* et *post-condition* ;
- le mot clé **self** sert à identifier l'instance contextuelle à laquelle la contrainte se rapporte ;
- à la place du mot clé **self**, il est possible d'inscrire un nom d'instance ;
- il est possible de nommer explicitement une contrainte afin de pouvoir la désigner par la suite. Le nom de la contrainte est placé après le type de contrainte (**inv**, **pre** ou **post**) et avant le " :".
- les *stéréotypes* permettent d'étendre la sémantique des éléments de modélisation : il s'agit d'un mécanisme d'extensibilité du méta-modèle d'UML.

### 4.7.2 Les types de base et les opérateurs

Les types de base OCL sont les entiers, les booléens, les chaînes de caractères et les réels. Le type **OclAny** est également défini dans OCL. Il s'agit d'un super-type pour tous les types (de base et du modèle). Les opérateurs définis pour ce super-type sont hérités par tous les types. Les opérateurs suivants s'appliquent à ces types de base :

- opérateurs logiques : $>$, $<$, $\geqslant$, $\leq$, $\neq$, **and, or, xor, implies, if...then ...else... endif, not** ;
- opérateurs mathématiques : *+, -, *, /;*
- opérateurs pour les chaînes de caractères : **Concat, toUpper**.

Les types énumérés sont également autorisés dans les expressions OCL. La valeur d'un type énuméré est identifié par l'expression *nom_enumération*::*nom_énuméré*. OCL est un langage typé : une expression est dite valide lorsque les types de cette expression sont compatibles. OCL définit par contre une conformité de type entre un entier et un réel. L'opérateur **if...then...else...endif** définit une condition alternative. La clause **else** est obligatoire et introduit une expression de même type que l'expression de la clause **then**.

### 4.7.3 Les types et sous-types de classificateurs

Les classificateurs (classes, associations, ...) spécifient des types susceptibles de figurer dans l'expression de contraintes OCL. La conformité des types doit être vérifiée pour valider les expressions OCL définies. Les relations de généralisation entre classes permettent de comparer des types et leurs sous-types, puisque les sous-types se conforment à leurs super-types.

### 4.7.4 Les propriétés des objets

Les expressions OCL peuvent utiliser des instances de classificateurs ou des instances de type de base. Dans le cas des instances de classificateurs, il est possible d'employer des attributs, des sélecteurs et des extrémités de liens dans les contraintes OCL. L'évaluation de la valeur d'un attribut ou d'une opération de consultation d'une instance est obtenue en mentionnant le nom de l'instance, suivi d'un point (ou d'une flèche dans le cas de collections) puis de l'attribut ou de l'opération :

- l'expression **pre** définit des conditions vérifiées avant l'exécution de l'opération ; l'expression **post** définit des conditions vérifiées après l'exécution de l'opération.
- Il est possible de désigner l'objet retourné par une opération à l'aide du mot **result**.

### 4.7.5 Les valeurs antérieures

La présence d'une contrainte stéréotypée *post-condition* impose parfois l'évaluation de la valeur antérieure d'une propriété. En effet, lors de l'évaluation d'une propriété (un attribut, une opération ou une extrémité de lien) dans une post-condition, la valeur évaluée est celle obtenue après l'exécution de l'opération. Pour désigner la valeur de la propriété avant l'exécution de l'opération, le terme **@pre** est post-fixé au nom de la propriété.

### 4.7.6 La navigation vers les extrémités d'association

Il est possible de naviguer le long des liens pour passer d'une classe à un autre élément de modélisation. L'instance de départ est spécifiée, suivie d'un point (ou d'une flèche dans le

cas de collections), puis du nom du rôle opposé. Lorsque le nom du rôle n'est pas présent, le nom de la classe à l'extrémité concernée est utilisé, lequel commence par une minuscule.

### 4.7.7 Les collections

Dans le cas d'une association, l'expression retourne une collection d'objets dont le nombre est borné par la multiplicité du rôle. Ces types de collections sont :

- **Collection** : il s'agit d'un type abstrait définissant des opérations de manipulation pour les sous-types concrets ;
- **Set** (obtenu par navigation d'une association) : il s'agit d'un ensemble (tel un ensemble mathématique) ;
- **Bag** (obtenu par des navigations de plusieurs associations) : il s'agit d'un panier à provisions (aussi appelé parfois multi-ensemble), où un même élément peut être présent plusieurs fois ;
- **Sequence** (obtenu par une navigation vers une extrémité, avec la contrainte {**ordonnée**}) : il s'agit d'un panier à provisions, mais ordonné.

OCL définit des opérations pour valider, itérer sur ou consulter des éléments de collections. Voici une liste des opérations les plus couramment utilisées :

- **isEmpty()** : retourne une valeur booléenne ; cette valeur vaut vrai si l'ensemble est vide, faux sinon.
- **notEmpty()** : retourne la valeur vraie si l'ensemble contient au moins une instance, faux sinon.
- **size()** : retourne la cardinalité d'un ensemble (valeur de type entier).
- **count(*unObjet*)** : retourne un entier qui correspond au nombre de fois où *unObjet* figure dans la collection (pour un ensemble, le résultat de cette opération est forcément inférieur ou égal à 1).
- **sum()** : retourne la valeur de l'addition de tous les éléments de la collection.
- **select(...)** : restriction en fonction du critère précisé entre parenthèses. Cette opération est évaluée à un sous-ensemble de la collection de départ.
- **collect(*uneExpression*)** : la collection obtenue est un bag ou un multi-ensemble qui est construite à partir de la collection de départ, mais qui contient des éléments d'un autre type (pas forcément un sous-ensemble).
- **includes(*unObjet*)** : retourne la valeur vraie si *unObjet* est un élément de la collection, faux sinon.
- **forAll(*uneExpression*)** : retourne la valeur vraie si *uneExpression* est vraie pour tous les éléments de la collection, faux sinon.
- **exists(*uneExpression*)** : retourne la valeur vrai si *uneExpression* est vraie pour au moins un élément de la collection, faux sinon.
- **excluding(*unObjet*)** : retourne toutes les occurrences de l'ensemble excepté celle de l'objet *unObjet*.
- **including(*unObjet*)** : retourne toutes les occurrences de l'ensemble plus celle de l'objet *unObjet*.
- **excludes(*unObjet*)** : retourne la valeur vraie si *unObjet* n'est pas un élément de la collection d'où l'expression est appelée sinon faux.

– **union(*set*)** : retourne l'union de l'ensemble depuis lequel l'expression est appelée plus les occurrences de l'ensemble donné en paramètre.

### 4.7.8 Les classes-associations

La navigation vers des classes-associations est définie par le nom de l'instance d'un des classificateurs reliés par l'association concernée, suivi d'un point, puis du nom de la classe-association commençant par une minuscule.

La navigation d'une classe-association vers les instances qui participent à l'association est également possible. Dans ce cas, le nom de l'instance de la classe-association est suivi d'un point, puis du nom du rôle concerné.

### 4.7.9 Les variables

Lorsqu'une expression apparaît plusieurs fois dans une condition, il n'est pas nécessaire de la réécrire plusieurs fois et une variable peut être définie. Les mots clés **let** et **in** introduisent la définition d'une variable sous la forme :

(**let** *variable : type = expression*)* **in** *expr*

## 4.8 Conclusion

Nous avons présenté dans ce chapitre une vue générale des diagrammes UML que nous allons traiter dans l'extraction de vues UML depuis une spécification B. Les notations objets comme UML sont des techniques très performantes pour la modélisation de systèmes. Elles mettent en valeur les deux aspects de la modélisation qui sont la partie dynamique et la partie statique. Les modèles à objets sont faciles à comprendre car directement interprétables dans le monde réel et surtout beaucoup moins abstraits que les modèles mathématiques. Ils montrent une vue d'ensemble bien structurée du système. Néanmoins UML souffre d'un manque de formalisme. Ses créateurs ont essayé de le combler par la création du langage OCL qui permet de formaliser les contraintes sans atteindre le niveau de formalisme des notations mathématiques. Malgré tout UML reste la notation standard la plus répandue car elle est adaptée à tous les types de développement. De plus elle est supportée par de nombreux outils.

# Chapitre 5

# Etat de l'art : Dérivation d'UML vers B

## 5.1 Introduction

Les deux chapitres précédents exposent les caractéristiques de la méthode B et de la notation UML. Le but de ce chapitre est de présenter un état de l'art des travaux publiés concernant l'intégration des méthodes formelles et semi-formelles dans le développement logiciel. Les travaux traitant d'une extraction de vues semi-formelles depuis une spécification formelle sont peu nombreux [32, 7]. Les travaux dans le domaine consistent principalement en une dérivation de notations semi-formelles vers une spécification formelle telle que B [61, 69]. On peut citer aussi d'autres travaux similaires vers Z :

- Dupuy, Ledru et Chabre-Peccoud ont réalisé une dérivation de OMT vers Object-Z [22, 23].
- France, Bruel, Larrondo-Petrie et Grant ont défini une dérivation depuis la méthode fusion [13] vers Z [42].
- Shroff et France ont dérivé UML vers Z [86].

Ce chapitre rappelle les points essentiels des thèses de Nguyen [77], Meyer [73] et Ledang [64]. Les travaux de Nguyen ont posé les premiers concepts synthétiques et théoriques de la dérivation de méthodes semi-formelles (OMT et UML) pour les diagrammes de classes et d'états-transitions vers B. Par la suite les travaux de Meyer ont approfondi la dérivation des diagrammes de classes et d'états-transitions depuis les diagrammes UML. Ledang a réalisé une extension des travaux de Meyer en y ajoutant la dérivation des diagrammes de collaboration et de cas d'utilisation. D'autre part Sekerinski a aussi réalisé des travaux sur le domaine de traduction des statecharts vers les machines abstraites [84]. D'autres travaux, qui ne sont pas en relation avec l'utilisation conjointe des deux approches, traitent de l'extraction de systèmes de transitions étiquetées (automates) depuis une spécification B [16].

## 5.2 Dérivation à partir de diagrammes de classes UML

Les diagrammes de classes sont les plus utilisés pour modéliser la partie statique d'un système. Ils comprennent des éléments comme les classes, les attributs, les associations et la généralisation. La dérivation de ces diagrammes est insérée dans les données des machines abstraites.

### 5.2.1 Dérivation des classes

Selon Meyer une classe *Classe* (fig 5.1) est dérivée formellement en B par la construction d'une machine abstraite *Classe* (fig 5.2). Celle-ci déclare :

- l'ensemble *CLASSE* des objets possibles de *Classe* : *CLASSE* est une constante définie comme un sous-ensemble de l'ensemble de tous les objets possibles *OBJETS*,
- l'ensemble *classe* des objets instanciés : *classe* est défini comme un sous-ensemble de *CLASSE* et initialisé par l'ensemble vide.

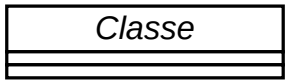


FIG. 5.1 – Une classe en UML

| | |
|---|---|
| MACHINE | *Classe* |
| CONSTANTS | *CLASSE* |
| PROPERTIES | $CLASSE \subseteq OBJETS$ |
| VARIABLES | *classe* |
| INVARIANT | $classe \subseteq CLASSE$ |
| INITIALISATION | $classe := \emptyset$ |
| END | |

FIG. 5.2 – Dérivation de la classe en B

Une classe peut être paramétrée, elle sera appelée classe *template*. Ceci permet de donner un paramètre et son typage à une classe (fig 5.3).

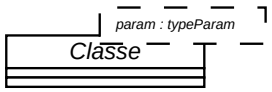


FIG. 5.3 – Une classe *template*

Une classe *template* est dérivée formellement en B par la construction d'une machine abstraite paramétrée *Classe*. Cette machine est la même que dans la dérivation précédente et possède en plus un paramètre typé dans la clause CONSTRAINTS (fig 5.4).

| | |
|---|---|
| MACHINE | *Classe(param)* |
| CONSTRAINTS | $param \in typeParam$ |
| ... | |

FIG. 5.4 – Dérivation d'une classe *template*

### 5.2.2 Attribut

Les attributs ont un type en UML et plusieurs visibilités possibles. La dérivation formelle en B de l'attribut ne peut être réalisée qu'avec une visibilité privée car elle est la seule à posséder un équivalent en B. Le typage de l'attribut et l'appartenance à l'objet instancié de la classe *Classe* (fig 5.5) sont définis par la relation *attrib* (fig 5.6) typée de la manière suivante dans l'invariant : $attrib \in classe \leftrightarrow typeAttrib$, attrib étant le nom de l'attribut.

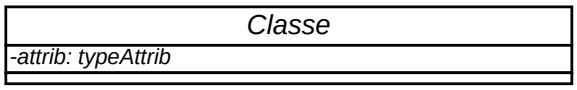


FIG. 5.5 – Un attribut

...
VARIABLES $attrib, classe$
INVARIANT $attrib \in classe \leftrightarrow typeAttrib$
INITIALISATION $attrib := \emptyset$
...

FIG. 5.6 – Dérivation d'un attribut

### 5.2.3 Opération

#### 5.2.3.1 Opération standard

Dans une classe *Classe* une méthode UML est dérivée formellement en B dans la machine *Classe* par la construction partielle d'une opération (fig 5.8) car le diagramme de classes (fig 5.7) ne donne pas le contenu (code) de l'opération. En effet le résultat de la dérivation donne pour les opérations les informations suivantes :

- le nom de l'opération,
- le nom et le type des paramètres d'entrée,
- le type des paramètres de sortie.

Durant la dérivation, il faut nommer les paramètres de sortie arbitrairement. D'autre part, la machine représente la classe dont il faut définir l'instance de classe concernée par l'exécution de la méthode. On est donc obligé de modifier le prototype de la méthode pour y faire apparaître un paramètre supplémentaire *oo* qui correspond à l'identifiant de l'objet. Cet identifiant est contenu dans la variable ensemble *classe* d'où son typage dans les pré-conditions $oo \in classe$.

#### 5.2.3.2 Le constructeur

Le constructeur que l'on nomme *creerClasse* (fig 5.9) devient une opération commune de la machine *Classe* (fig 5.10). Une nouvelle instance de classe *oo* est créée lorsque la valeur *oo* est retournée par l'opération *creerClasse* et sa valeur *new* est définie par $new \in CLASSE$ -

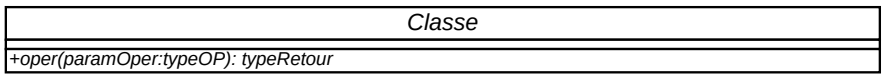


FIG. 5.7 – Une méthode

...
OPERATIONS
*out*←*oper(oo,paramOper)*≙
PRE
*oo* ∈ *classe* ∧
*paramOper* ∈ *typeOP*
THEN
*out* :∈ *typeRetour*
END;
...

FIG. 5.8 – Dérivation d'une méthode

*classe*. L'opération vérifie au préalable dans sa pré-condition que l'ensemble *CLASSE - classe* ne soit pas vide. Les attributs sont initialisés. Il existe deux possibilités. La valeur initiale de l'attribut est explicitement définie ; on utilise cette valeur *val1* pour augmenter la relation *attrib1* du couple *new* ↦ *val1*. La valeur est définie par un paramètre *valAttrib2* typé dans les pré-conditions de l'opération dérivée du constructeur avec le type associé de la variable *valAttrib2* ∈ *typeAttrib2* pour augmenter la relation *attrib2* par le couple *new* ↦ *valAttrib2*.

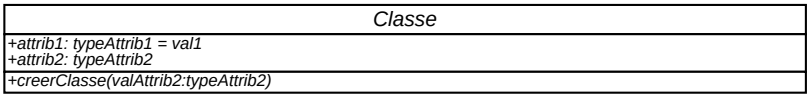


FIG. 5.9 – Un constructeur

### 5.2.4 Association

Les associations sont des liens entre deux classes *Class1* et *Class2* (fig 5.11). Elles sont dérivées formellement en B par une relation binaire (fig 5.12) entre les variables B *classe1* et *classe2*. On remarque que même si le sens est facultatif en UML il faut en spécifier un pour effectuer la dérivation car il existe obligatoirement en B de même que le nom est nécessaire pour nommer la relation issue de la dérivation.

Cette relation peut être raffinée selon la multiplicité ou d'autres contraintes telles que l'agrégation. Les multiplicités d'une association peuvent être dérivées formellement en B selon le tableau 5.1. Dans ce tableau les entiers $n$ et $m$ sont définis par $n \geq 2$ et $m \geq n$.

...
OPERATIONS
*oo*←*creerClasse(valAttrib2)*$\widehat{=}$
PRE
*not(CLASSE-classe* = $\emptyset$ *)*$\wedge$
*valAttrib2* $\in$ *typeAttrib2*
THEN
ANY *new* WHERE
*new* $\in$ *CLASSE-classe*
THEN
*classe* := *classe* $\cup$ {*new*} ||
*attrib1* := *attrib1* $\cup$ {*new* $\mapsto$ *val1*} ||
*attrib2* := *attrib2* $\cup$ {*new* $\mapsto$ *valAttrib2*} ||
*oo* := *new*
END
END ;
...

FIG. 5.10 – Dérivation du constructeur

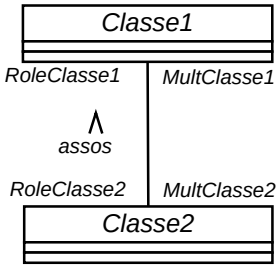


FIG. 5.11 – Une association

...
VARIABLES *assos, classe1, classe2*
INVARIANT *assos* $\in$ *classe1* $\leftrightarrow$ *classe2*
INITIALISATION *assos* := $\emptyset$
...

FIG. 5.12 – Dérivation d'une association

### 5.2.5 Généralisation

La généralisation d'une sous-classe *SousClasse* par une classe *SuperClasse* est dérivée formellement en B en appliquant pour la formalisation de *SuperClasse* le schéma de dérivation d'une classe ordinaire et en redéfinissant celui de la sous-classe *SousClasse* de la manière suivante :

| Multiplicité | MultClasse1 | MultClasse2 |
|---|---|---|
| * ou 0..* | $assos \in classe1 \leftrightarrow classe2$ | $assos^{-1} \in classe2 \leftrightarrow classe1$ |
| 0..1 | $assos \in classe1 \nrightarrow classe2$ | $assos^{-1} \in classe2 \nrightarrow classe1$ |
| 1 | $assos \in classe1 \rightarrow classe2$ | $assos^{-1} \in classe2 \rightarrow classe1$ |
| 1..* | $assos \in classe1 \leftrightarrow classe2 \wedge$ $dom(assos) = classe1$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $ran(assos) = classe2$ |
| n | $assos \in classe1 \leftrightarrow classe2 \wedge$ $\forall x.(x \in classe1 \Rightarrow card(assos[x]) = n)$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $\forall x.(x \in classe2 \Rightarrow card(assos[x]) = n)$ |
| 0..n | $assos \in classe1 \leftrightarrow classe2 \wedge$ $\forall x.(x \in classe1 \Rightarrow card(assos[x]) \leq n)$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $\forall x.(x \in classe2 \Rightarrow card(assos[x]) \leq n)$ |
| 1..n | $assos \in classe1 \leftrightarrow classe2 \wedge$ $\forall x.(x \in classe1 \Rightarrow card(assos[x]) \leq n) \wedge$ $dom(assos) = classe1$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $\forall x.(x \in classe2 \Rightarrow card(assos[x]) \leq n) \wedge$ $ran(assos) = classe2$ |
| n..* | $assos \in classe1 \leftrightarrow classe2 \wedge$ $\forall x.(x \in classe1 \Rightarrow card(assos[x]) \geq n)$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $\forall x.(x \in classe2 \Rightarrow card(assos[x]) \geq n)$ |
| n..m | $assos \in classe1 \leftrightarrow classe2 \wedge$ $\forall x.(x \in classe1 \Rightarrow n \leq card(assos[x]) \leq m)$ | $assos^{-1} \in classe2 \leftrightarrow classe1 \wedge$ $\forall x.(x \in classe2 \Rightarrow n \leq card(assos[x]) \leq m)$ |

TAB. 5.1 – Affinement de la dérivation d'une association par ses multiplicités

- la machine *SousClasse* utilise la machine *SuperClasse* en insérant la machine *SuperClasse* dans la clause USES de la machine *SousClasse* ; On ne définit pas la constante *SOUS-CLASS* dans la machine *SousClasse* car on peut utiliser la constante *SUPERCLASS* de la machine *SuperClasse* pour la remplacer afin de modéliser l'ensemble des objets possibles de la classe *SousClasse*,
- l'ensemble des objets instanciés de *SousClasse* est un sous-ensemble *sousclasse* de celui de sa super-classe (*superclasse*),
- les attributs de *SousClasse* sont dérivés de la même façon que pour une classe ordinaire sachant que la clause USES permet l'accès aux attributs de la machine *SuperClasse*.

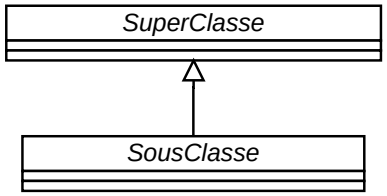


FIG. 5.13 – Une généralisation

## 5.3 Extension de la dérivation des diagrammes de classes

Ce paragraphe relate les travaux de Ledang [64] qui complètent les travaux de Meyer à propos de la dérivation des diagrammes de classes. Les travaux de Meyer ne prenaient pas en compte les problèmes dus aux appels d'opérations à l'intérieur de la même machine ce que Ledang réalise dans ses travaux.

| | |
|---|---|
| MACHINE | *SousClasse* |
| SEES | *SuperClasse* |
| VARIABLES | *sousclasse* |
| INVARIANT | *sousclasse* $\in$ *superclasse* |
| INITIALISATION | *sousclasse* := $\emptyset$ |
| END | |

FIG. 5.14 – Dérivation de la généralisation

### 5.3.1 Motivations

Du fait des différences marquées entre les propriétés du langage B et UML, on est obligé de modifier complètement l'organisation définie dans les vues UML pour la dériver en B. Meyer ne tient pas compte du fait qu'une méthode UML peut appeler une autre méthode de la même classe. Ceci se dérive par une opération qui appelle une autre opération de la même machine. On ne peut pas le réaliser en B car les appels mutuels peuvent faire apparaître des cycles sans fin. De plus il ne prend pas en compte les problèmes d'homonymie des attributs et des méthodes qui vont apparaître dans la même spécification. D'autre part, l'utilisation des diagrammes de réalisation (diagrammes d'interaction ou d'activité) vont permettre de dériver le corps (code) des opérations et donc les appels à d'autres opérations. Ces appels sont possibles en B à la condition qu'ils ne soient pas cycliques. Un cycle d'appel est une opération *op1* qui appelle *op2* qui appelle elle-même *op1*. En B, l'opérateur parallèle || utilisé dans les machines abstraites n'autorise pas l'appel de deux opérations de la même machine incluse (clause INCLUDES).

### 5.3.2 Dérivation des prototypes et appels d'opérations

Les prototypes d'opérations sont dérivés de la même manière que pour les travaux de Meyer décrits dans les paragraphes précédents. La différence réside dans le renommage de l'opération. En effet le nom de l'opération devient le nom de classe suivi du caractère '_' suivi du nom d'opération. Cette opération est similaire pour les noms d'attributs. Ceci a pour but d'éviter les conflits de nom sachant qu'il faut interdire le caractère '_' dans les noms de machines et opérations (exemples de problèmes : machine A_B avec une opération C ou machine A avec une opération B_C). Les appels des opérations peuvent être dérivés sous les conditions suivantes :

- les appels ne sont pas cycliques,
- le couple d'opération appelante/appelée ne fait pas partie de la même machine.

Il faut donc séparer l'opération appelée de l'opération appelante sans oublier par la même occasion de scinder les données. Dans le cas où ces données correspondent à deux opérations de deux machines différentes elle doivent être dupliquées dans les deux machines. Ensuite on crée un lien entre les deux machines qui possèdent l'opération appelante (*Machine1*) et appelée (*Machine2*) pour que l'appel soit possible. Au niveau des machines abstraites, on utilise la clause INCLUDES dans la machine *Machine1* pour permettre la visibilité et l'utilisation de l'opération appelée. L'opérateur parallèle || n'accepte pas deux appels d'opérations de la même machine par conséquent on les déplace dans le raffinement dont le code est basé sur l'opérateur

de séquencement ; qui l'accepte. Le lien entre les deux machines est aussi réalisé avec la clause INCLUDES. Au niveau de l'implémentation on importe la machine des opérations appelées avec la clause IMPORTS. Une autre contrainte est relative à l'instruction WHILE ... DO ...VARIANT ... INVARIANT ... END qui n'est permise que dans les implémentations. Il est donc nécessaire de remplacer la substitution WHILE par la substitution identité si aucune modification des données n'est nécessaire ou par une autre dans le cas contraire dans l'abstraction et les raffinements et de maintenir la substitution WHILE dans l'implémentation.

### 5.3.3 Classification et organisation des opérations

L'organisation des opérations s'effectue par couches. Ces couches sont au nombre de trois. Dans un premier temps on trie les opérations par groupe. Un groupe est constitué d'une opération appelante et de ses opérations appelées. Ensuite, connaissant les opérations appelantes, appelées et appelantes/appelées on les trie respectivement de la manière suivante :

- Première couche : les opérations appelantes uniquement,
- Deuxième couche : les opérations appelées/appelantes,
- Troisième couche : les opérations appelées uniquement.

A ce stade il reste un problème. La machine correspondant à la troisième couche importée dans la machine de la première couche est aussi importée dans la deuxième couche qui est elle-même importée dans la première couche. En effet les opérations de la première et de la deuxième couches appellent les opérations de la troisième mais ce n'est pas possible. Par conséquent on importe la machine de la troisième couche dans celle de la deuxième puis on importe celle de la deuxième dans la première. Dans le but de relayer l'appel de la première couche vers la troisième on utilise des opérations fictives dans la deuxième couche avec le suffixe *dum* qui lance l'appel de la véritable opération dans la troisième couche.

## 5.4 Dérivation à partir de diagrammes d'états-transitions

Les machines à états finis à travers les diagrammes d'états-transitions ou statecharts [50] sont utilisées depuis longtemps par diverses méthodes. Les méthodes semi-formelles du type d'UML ont intégré dans la partie dynamique ce type de diagramme permettant de modéliser le comportement d'un objet. Il existe deux dérivations formelles différentes des diagrammes d'états-transitions en B. La première est la traduction des statecharts vers les machines abstraites dans [84]. La deuxième est parallèle à la dérivation d'un diagramme de classes [73] présenté dans le paragraphe précédent. Le seul point commun aux deux versions de dérivation est les états représentés par des éléments d'ensembles énumérés. Ledang [64] a réalisé une extension des travaux de Meyer en spécifiant à travers la dérivation les opérations décrites dans le diagramme d'états-transitions. D'autre part certains travaux ont mis en évidence la possibilité de traduire une spécification B en un système de transitions étiquetés [16].

### 5.4.1 Dérivation de statecharts en machines abstraites

Dans cette partie un résumé des méthodes utilisées par Sekerinski [84] et Meyer [73] est exposé. Les deux méthodes sont identiques dans le principe. La différence réside dans le fait que la dérivation décrite par Meyer est attachée à un diagramme de classes dont la dérivation est définie dans la partie précédente.

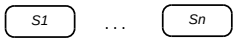


FIG. 5.15 – Des états

#### 5.4.1.1 Les états

Les états du système sont symbolisés par un carré aux angles arrondis (fig 5.15). Ils sont nommés de *S1* à *Sn*.

| | |
|---|---|
| ... | |
| SETS | $S=\{S1,...,Sn\}$ |
| VARIABLES | $s$ |
| INVARIANT | $s \in S$ |
| ... | |

FIG. 5.16 – Dérivation des états selon Sekerinski

Pour Sekerinski ils deviennent les éléments de l'ensemble énuméré S. Ensuite il suffit de créer une variable pour définir l'état courant du système. L'ensemble énuméré est utilisé pour le typage de variables scalaires.

| | |
|---|---|
| . . . | |
| SETS | $CLASS\_STATE = \{S1, \ldots, Sn\}$ |
| VARIABLES | $class\_state$ |
| INVARIANT | $class\_state \in Class \rightarrow CLASS\_STATE$ |
| INITIALISATION | $class\_state := \emptyset$ |
| ... | |

FIG. 5.17 – Dérivation des états selon Meyer

Meyer utilise une fonction totale *classe_state* dont le domaine correspond à l'identifiant d'objet (fig 5.17). Les autres différences entre les deux méthodes de dérivation sont minimes et résident dans le nommage des différentes entités utilisées. Par exemple, l'ensemble énuméré *S* (Sekerinski) devient *CLASSE_STATE* (Meyer).

#### 5.4.1.2 L'initialisation

L'initialisation d'un diagramme d'états-transitions est représenté par un disque noir plein et une flèche dirigée vers l'état cible. L'initialisation peut être réalisée de deux manières différentes, comme dans les deux exemples suivants :

- déterministe : il existe un seul état cible (fig 5.18) ;
- non déterministe : il existe plusieurs états cible (fig 5.19).

Pour Sekerinski la traduction de l'initialisation (fig 5.20) est réalisée par une affectation ( $:=$) dans le cas où elle est déterministe ou par l'opérateur *devient tel que* ( $:\in$) quand elle est non déterministe.

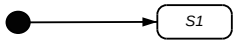


FIG. 5.18 – Initialisation déterministe

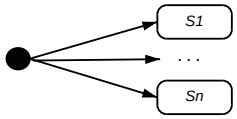


FIG. 5.19 – Initialisation non déterministe

...
INITIALISATION $(s := S1 \mid s :\in S)$
...

FIG. 5.20 – Dérivation de l'initialisation selon Sekerinski

Pour Meyer la dérivation est la même modulo les différences décrites dans le paragraphe précédent (fig 5.21). Aussi Meyer utilise un constructeur nommé *creerClasse* qui détermine l'identifiant d'objet utilisé dans la fonction totale *classe_state*.

```
...
OPERATIONS
    oo←creerClasse≙
    PRE
        not(CLASSE - classe = ∅)
    THEN
        ANY new WHERE
            new ∈ CLASSE - classe
        THEN
            ...
            classe_state(new) := S1 ||
            oo := new
        END
    END
...
```

FIG. 5.21 – Dérivation de l'initialisation selon Meyer

#### 5.4.1.3 Les transitions

Les transitions (fig 5.22) sont représentées par une flèche partant d'un état source et dirigé vers un état cible.

Pour Sekerinski dans la dérivation (fig 5.23) l'état source est défini par une condition élaborée à l'aide des substitutions IF, CASE ou SELECT. L'opération prend le nom de la

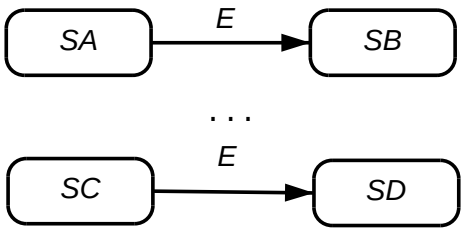


FIG. 5.22 – Des transitions

transition et la valeur affectée à la variable $s$ dans l'exemple suivant est définie par l'état cible de la transition.

```
...
OPERATIONS
    E ≙
      (IF s=SA THEN s :=SB END ... ELSIF s=SC THEN s :=SD |
      SELECT s=SA THEN s :=SB END ... WHEN s=SC THEN s :=SD |
      CASE s OF EITHER SA THEN s :=SB END ... OR SC THEN s :=SD)
    END
...
```

FIG. 5.23 – Dérivation des transitions selon Sekerinski

Pour Meyer les principes sont les mêmes excepté l'affectation qui est opérée sur la fonction totale *classe_state* (fig 5.24 ). De plus il génère une opération par transition nommé par la chaîne de caractères *Transition* suivie des noms des états source et cible de la transition. Dans l'exemple de la figure 5.22 la transition traduite part de l'état *SA* et se dirige vers l'état *SB*.

```
...
OPERATIONS
    TransitionAB ( oo )≙
    PRE
        oo ∈ class ∧
        class_state ( oo ) = SA
    THEN
        class_state ( oo ) := SB
    END;
...
```

FIG. 5.24 – Dérivation d'une transition selon Meyer

#### 5.4.1.4 Paramètres conditions et actions additionnelles

Pour Sekerinski les transitions peuvent être agrémentées de paramètres (*para*) condition (*cond*) ou d'action (*act*) qui sont extérieurs à la compréhension du diagramme d'états-transitions (fig 5.25). On constate que ces éléments sont observés sur le diagramme d'états-transitions. En effet le paramètre est parenthésé à la suite du nom de l'événement suivi de la condition externe entre crochets qui ne concerne pas les variables du diagramme d'états-transitions. Ensuite le caractère '/' précède les actions externes qui ne concernent pas non plus les variables du diagramme d'états-transitions.

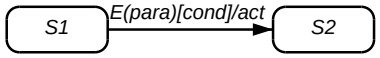


FIG. 5.25 – Une transition complexe

Dans la traduction, le paramètre apparaît à la suite du nom de l'événement. La condition est concaténée à celle de l'état source et l'action à l'affectation relative à l'état cible (fig 5.26).

...
OPERATIONS
*E(para)*$\widehat{=}$
(IF|SELECT) $s = S1 \wedge cond$
THEN $s := S1' \parallel act$ END
...
END
...

FIG. 5.26 – Dérivation d'une transition complexe

### 5.4.2 Extension de la dérivation associée au diagramme de classes

Ce paragraphe traite les travaux de Ledang sur l'extension de la dérivation des diagrammes d'états-transitions en spécification B. Cette extension a été illustrée par l'exemple de l'ascenseur cité ci-après.

#### 5.4.2.1 Exemple de l'ascenseur

Les ascenseurs desservent les étages d'un bâtiment. Chaque étage possède un bouton d'appel de l'ascenseur. A l'intérieur de l'ascenseur, il y a un bouton par étage pour choisir sa destination. Quand on appuie sur ce bouton il reste allumé jusqu'à l'arrivée de l'ascenseur à l'étage concerné. Un voyant par étage détermine l'étage courant. Lorsque l'ascenseur est en mouvement et reçoit un autre appel, il change d'étage de destination une fois le premier appel réalisé par son arrivée (l'ouverture et la fermeture des portes). L'ascenseur est mis en attente au dernier étage demandé lorsque toutes les demandes ont été satisfaites. Une demande de l'étage courant de l'ascenseur provoque l'ouverture des portes puis peu après sa fermeture. Dans le diagramme de classes suivant (fig 5.27) la classe *Lift* correspond à

l'ascenseur et *Button* aux boutons. Les types *DIRECTION, STATUS* et *FLOOR* sont définis par les ensembles respectifs suivants {*up,down*}, {*opened,closed*} et {*ground..top*}.

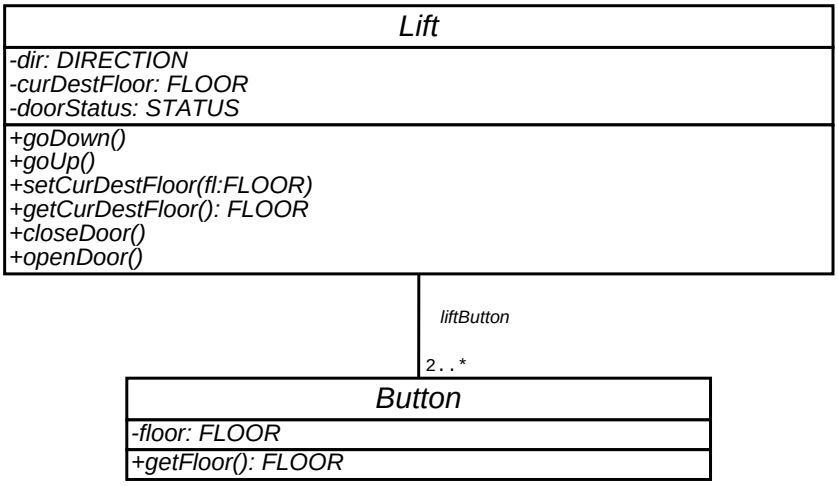


FIG. 5.27 – Diagramme de classes de l'ascenseur

La figure 5.28 montre le diagramme d'états-transitions du bouton. Lorsqu'on appuie sur le bouton il passe à l'état *on* (allumé) et il s'éteint à l'état *off*.

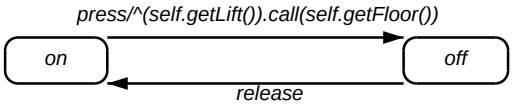


FIG. 5.28 – Diagramme d'états-transitions du bouton

La figure 5.29 représente le diagramme d'états-transitions de l'ascenseur. Quand l'ascenseur est en mouvement pour un premier appel (call) et qu'un deuxième appel survient, il est différé car l'ascenseur répondra au premier appel avant le deuxième. On parle de communication asynchrone. L'argument *fl* de l'appel call est le numéro d'étage d'où il provient. Au contraire de l'appel (call) l'extinction du bouton release est synchrone car il est instantané par rapport à l'arrivée de l'ascenseur à l'étage demandé.

#### 5.4.2.2 Raffinement et implémentation des transitions et actions

Les travaux de Ledang dérivent les états et transitions de la même manière que Meyer, le cas général est présenté dans la figure 5.32, et complètent, si nécessaire, le projet B par une machine correspondant à la console de contrôle du système et par une autre opération dérivée dans cette nouvelle machine (fig 5.30) :

- un argument d'opération pour identifier l'objet concerné : *ll*
- une condition qui définit l'état source : *lift_State(ll) = lift_visit*
- une affectation qui définit l'état cible : *lift_State(ll) := lift_ready*

Le nom de l'opération (fig 5.30) utilisé est celui de la transition du diagramme d'états-transitions (fig 5.29) correspondant et précédé du nom de la classe et du caractère '_' pour éviter les problèmes d'homonymie. Les classes sont dérivées dans la même machine.

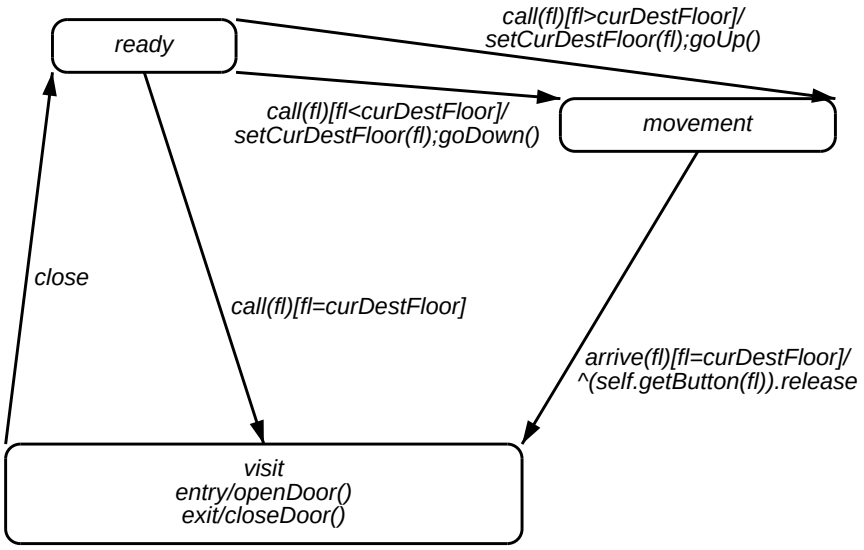


FIG. 5.29 – Diagramme d'états-transitions de l'ascenseur

MACHINE
*System*
SEES
*Types*
...
VARIABLES
*lift, lift_dir, lift_curDestFloor, lift_doorStatus, lift_state, button ...*
INVARIANT
$lift \subseteq LIFT \wedge$
$lift\_dir \in lift \rightarrow DIRECTION \wedge ...$
OPERATIONS
*lift_close ( ll )*$\widehat{=}$
PRE
$ll \in lift \wedge$
THEN
SELECT *lift_State(ll) = lift_visit* THEN
*lift_doorStatus(ll) := status_closed* ||
*lift_State(ll) := lift_ready*
ELSE *skip* END
END ;
...
END

FIG. 5.30 – Spécification de la partie *system* de l'ascenseur

Au contraire de Meyer les actions d'entrée et de sortie d'états sont prises en compte. En effet l'action de sortie de l'état source et celle d'entrée de l'état cible sont dérivées dans

MACHINE
*Types*
SETS
*OBJECTS ;*
*DIRECTION = {direction_up,direction_down} ;*
*STATUS = {status_opened,status_closed} ;*
*BUTTON_STATE = {button_on,button_off} ;*
*LIFT_STATE = {lift_ready,lift_movement,lift_visit}*
CONSTANTS
*floor_ground, floor_top, LIFT, BUTTON*
PROPERTIES
$floor\_ground \in NAT \wedge$
$floor\_top \in NAT \wedge$
$floor\_ground < floor\_top \wedge$
$LIFT \subseteq OBJECTS \wedge$
$BUTTON \subseteq OBJECTS \wedge$
END

FIG. 5.31 – Spécification de la machine *Types* de l'ascenseur

l'opération qui correspond à la transition. Dans l'exemple de l'ascenseur la dérivation de la transition *close* contient l'action de sortie de l'état *visit*, *exit/closeDoor()*. Elle est dérivée formellement en B par *lift_doorStatus(ll) := status_closed*. Les types sont utilisés par toutes les machines du projet B donc ils sont définis dans la machine *TYPES* (fig 5.31 et 5.34) qui est vues (clause *SEES*) par toutes celles qui utilisent ses types.

La machine *System* (fig 5.34) a été créée dans le but de pouvoir appeler des opérations de la machine *Lift* car une opération de la machine *lift* ne peut pas appeler une opération de cette même machine. Aussi on ne peut pas utiliser les opérations de la machine *Lift* (fig 5.32) dans la machine abstraite *System* car la substitution || n'accepte pas l'appel de deux opérations de la même machine en simultané. On est donc obligé de créer le raffinement de la machine *System* (fig 5.33) pour utiliser les opérations de la machine *Lift* (fig 5.32).

On remarque l'apparition de l'opération *lift_isVisit* qui est nécessaire dans l'implémentation *System_imp* pour vérifier l'état source avant la transition de *visit* vers *ready*.

#### 5.4.2.3 Communication entre diagrammes d'états-transitions

La communication entre diagrammes d'états-transitions est réalisée de deux manières différentes :

- synchrone
- asynchrone

La première est réalisée par un simple appel de l'opération dans l'opération dérivée de la transition qui demande la synchronisation. La deuxième est différée en attendant que la transition qui doit être exécutée puisse l'être. Par conséquent on utilise une file d'attente qui mémorise les appels d'exécution Dans l'exemple de l'ascenseur, l'appel release est synchrone tandis que l'appel call lui est asynchrone. En effet, selon le cahier des charges établi dans

MACHINE
*Lift*
SEES
*Types*
...
VARIABLES
*lift, lift_dir, lift_curDestFloor, lift_doorStatus, lift_state ...*
INVARIANT
$lift \subseteq LIFT$
...
DEFINITIONS
$FLOOR == (floor\_ground .. floor\_top)$
OPERATIONS
$lift\_transVisitReady(ll) \widehat{=}$
PRE
$ll \in lift \wedge$
$lift\_State(ll) = lift\_visit$
THEN
$lift\_State(ll) := lift\_ready$
END;
$lift\_closeDoor(ll) \widehat{=}$
PRE
$ll \in lift$
THEN
$lift\_doorStatus(ll) := status\_closed$
END;
$bb \leftarrow lift\_isVisit(ll) \widehat{=}$
PRE
$ll \in lift$
THEN
$bb := BOOL(lift\_State(ll) = lift\_visit)$
END
...
END

FIG. 5.32 – Spécification de l'ascenseur : cas général (travaux de Meyer)

les travaux de Ledang [64], si l'ascenseur est déjà appelé, un autre appel ne sera pas pris en considération avant que le premier ait abouti. On crée donc une file d'attente définie par le typage suivant $call\_queue \subseteq lift \times FLOOR$. Ainsi on connaît en permanence les appels de l'ascenseur en attente. On gère cette file d'attente à l'aide d'opérations d'ajout et de suppression des éléments.

```
IMPLEMENTATION
    System_imp
SEES
    Types
IMPORTS
    Lift
OPERATIONS
    lift_close(ll)≙
    VAR bb IN
        bb←lift_isVisit(ll) ;
        IF bb = TRUE THEN
            lift_closeDoor(ll) ;
            lift_transVisitReady(ll) ;
        END
    END;
...
END
```

FIG. 5.33 – Implémentation de la machine *system*

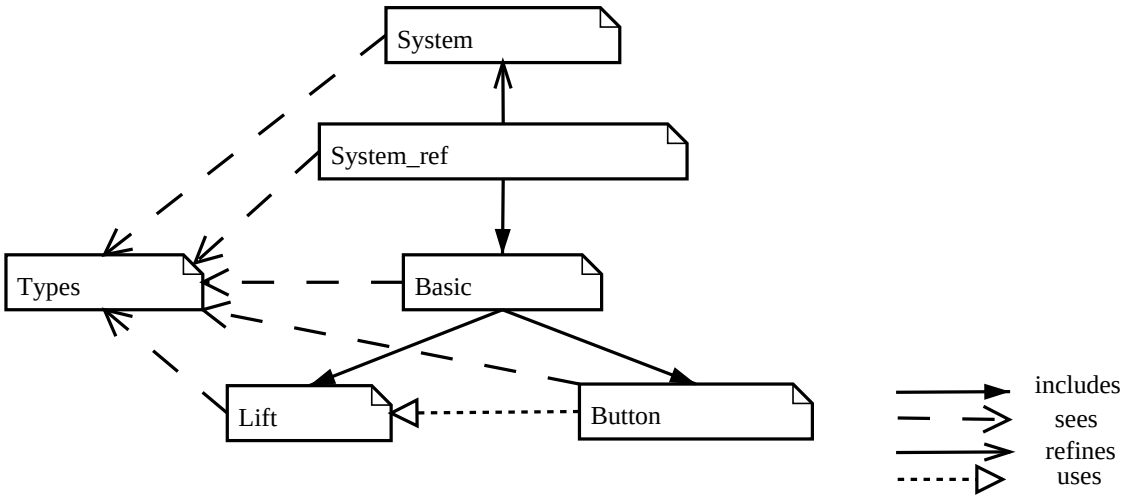


FIG. 5.34 – Architecture de la spécification B du système de contrôle d'ascenseurs

## 5.5 Traduction des expressions OCL en B

La spécification B dérivée doit être complétée par les corps d'opération et éventuellement par quelques propriétés de l'invariant dont les diagrammes UML ne peuvent exprimer la totalité. Le langage OCL, qui fait partie intégrante d'UML, permet de pallier les manques d'expressivité des diagrammes UML car il permet d'exprimer les invariants de classes et les pré et post-conditions des opérations.

### 5.5.1 Types de base prédéfinis OCL

Les expressions relatives à un type de base défini en OCL, sont dérivés en B par un type et ses opérateurs correspondants en langage B, suivant le travail de thèse de Ledang [64] (il existe d'autres travaux [36]), à l'exception du type réel. Pour Ledang, la solution est d'approximer une valeur réelle par une fraction définie par un couple de valeurs entières, par exemple la valeur *0,3333...* devient le couple $\{1 \mapsto 3\}$. D'autre part l'utilisation de l'opérateur de comparaison = donne lieu à quelques changements de la modélisation lorsqu'il est utilisé avec des fractions comme $\frac{a}{b} = \frac{c}{d}$ qui devient $a \times d = b \times c$.

### 5.5.2 Type collection

Le type collection, les opérations OCL prédéfinies et les opérations OCL de requête sur ces types sont formalisés en B différemment, suivant qu'on utilise le type Set, Bag ou Sequence. Soit t un type OCL et T sa formalisation en B :

- le type collection ***Set(t)***, qui désigne tous les sous-ensembles de ***t***, est modélisé en B par $\mathbb{P}(T)$ ;
- le type collection ***Bag (t)***, qui désigne tous les multi-ensembles avec les éléments de ***t***, est modélisé en B par $T \nrightarrow N$ ;
- le type collection ***Sequence (t)***, qui désigne toutes les suites d'éléments de ***t***, est modélisé directement en B par *seq(T)*.

### 5.5.3 Expression let ... in ...

L'expression let ... in ... en OCL permet de déclarer des macros qui permettent de rendre plus lisibles ces expressions. Comme B prévoit une clause spéciale DEFINITION pour définir des macros, il est logique d'utiliser les clauses de définition pour modéliser les expressions let ... in ... comme l'indique la dérivation qui suit :
l'expression let *v1 : type1 = val1, . . . , vn : typen = valn* in *exp* des post-conditions de *oper* permet de définir les variables locales *v1 , . . . , vn* :

DEFINITIONS
*v1 == val1 ;*
*... ;*
*vn == valn ;*

où *v1*, ..., *vn*, *val1*, ... et *valn* désignent la formulation en B de *v1 , ..., vn, val1, ...* et *valn*.

#### 5.5.3.1 Concepts structurels

Un attribut *attr* de type *typeAttr* dans une classe *Class* est modélisé par une variable B *attr* définie par l'invariant : $attr \in class \leftrightarrow typeAttr$, où la variable *class* modélise l'ensemble d'instances effectives de ***Classe*** (cf 5.2.2) et *typeAttr* est défini comme un ensemble B pour modéliser ***typeAttr***. Une Association binaire ***Assos*** est définie par l'invariant $assos \in class \leftrightarrow class2$. S'il y a des qualificatifs éventuels $q_1 : Q_1, ..., q_n : Q_n$, ils sont modélisés de la même manière qu'un attribut, $q_1 \in class2 \leftrightarrow Q_1, ..., q_n \in class2 \leftrightarrow Q_n$. Nous ajoutons aussi un invariant liant *assos* et $q_1, ..., q_n \in (assos^{-1} \otimes q_1 \otimes ... \otimes q_n) \in class \times Q_1 \times ... \times Q_n$.

### 5.5.4 Expression de post-condition

La formalisation en B des expressions OCL de post-condition d'un concept comportemental se présente comme suit : soit une spécification OCL {*preCond* , *postCond*} désignant la pré-condition et la post-condition d'une opération UML *oper*. L'opération B *operB* $\widehat{=}$ PRE *P* THEN *S* END dérivée de *oper* (UML) est générée de la manière suivante :

- la signature et une partie de la pré-condition P pour typer les arguments éventuels de l'opération sont générées selon le paragraphe 5.4.1.3 partie dérivation selon Meyer ;
- si *preCond* n'est pas vraie (*true*) (*preCond* est vide, elle n'est pas spécifiée explicitement) alors la pré-condition *P* est composée des prédicats de typage et des prédicats dérivés à partir de *preCond* ; dans le cas contraire, la pré-condition *P* est seulement composée des prédicats de typage pour les arguments éventuels ;
- la substitution *S* est générée à partir de la partie *postCond*, elle est de la forme : $x :\in$ *(P)* (*devient tel que*) dans le corps de l'opération B de telle sorte que :
  - $x$ est une liste de variables B qui sont principalement dérivées des attributs et des associations mentionnées dans *postCond* ;
  - le prédicat *P* est dérivé principalement de *postCond* en utilisant les schémas de dérivation pour les concepts UML structurels 5.5.3.1.

En plus d'utiliser la substitution *devient tel que*, Ledang propose d'utiliser la substitution IF ... THEN ... END dans la modélisation en B de *postCond* en cas de présence de la construction OCL **if** *cond* **then** *exp1* **else** *exp2* **endif** dans *postCond*. La formalisation en B de cette expression donne lieu à une clause :

- IF *condB* THEN *subst(exp1)* ELSE *subst(exp2)* END, si *exp2* n'est pas une expression : **if ... then ... else ... endif**
- IF *condB* THEN subst(*exp1)* ELSIF *condB2* THEN subst(*exp21)* ... END, si expr2 est de la forme : **if** *cond2* **then** *expr21 ...*

où *condB* et *condB2* sont les formalisations en B de *cond* et *cond2* (UML) et où *subst(expX)* désigne les substitutions B dérivées à partir de l'expression OCL *expX* où X est un nombre entier.

## 5.6 Extraction de systèmes de transitions étiquetées

Contrairement aux travaux exposés précédemment sur la dérivation, les travaux suivants s'apparentent à une extraction c'est-à-dire la transformation inverse de la dérivation. Ils n'utilisent pas une notation UML mais celle des systèmes de transitions étiquetées très proche de celle des diagrammes d'états-transitions UML. L'extraction est réalisée à partir de spécifications B vers des vues graphiques (Systèmes de transitions étiquetées).

### 5.6.1 Système de transitions étiquetées

Un système de transitions est défini par le quadruplet (N,M,L,$\mathcal{R}$) sachant que :

- N est l'ensemble des états,
- M est l'ensemble des états initiaux et M $\subset$ N,
- L est l'ensemble des étiquettes (labels) de transitions,

– $\mathcal{R} \in \mathbb{P}(\mathrm{N} \times \mathrm{L} \times \mathrm{N})$ est la relation de transition entre les états.

Une transition $(q_j, l, q_k)$ est aussi notée $q_j \xrightarrow{l} q_k$. Toutes les transitions étiquetées peuvent être réduites en une relation binaire sur les états (en oubliant les labels) $\mathrm{R} \in \mathbb{P}(\mathrm{N} \times \mathrm{N})$ par : $\mathrm{R} = \{(\mathrm{q},\mathrm{q'}) \mid \exists \mathrm{l}.(\mathrm{l} \in \mathrm{L} \wedge \mathrm{q} \xrightarrow{l} \mathrm{q'} \in \mathcal{R})\ \}$

La fermeture transitive est notée par $R^+$, la fermeture réflexive et transitive par $R^*$ et l'inverse par $R^{-1}$. Un état $\mathrm{q} \in \mathrm{N}$ est atteignable si $\mathrm{q} \in R^*[\mathrm{M}]$, autrement dit q appartient à l'image des états initiaux par la fermeture réflexive et transitive de R. Si $\mathrm{R} \subset N_1 \times N_2$ alors on note par $\mathrm{pre}[\mathrm{R}] \in \mathbb{P}(N_2) \rightarrow \mathbb{P}(N_1)$ la fonction des prédécesseurs sur l'ensemble des états et par $\mathrm{post}[\mathrm{R}] \in \mathbb{P}(N_1) \rightarrow \mathbb{P}(N_2)$ la fonction des successeurs. Soit Q un sous-ensemble de N, alors $\overline{Q}$ est le complément de Q par rapport à N. Les définitions suivantes sont habituellement utilisées sur des relations binaires [68, 87] où $\mathrm{R} \subset N_1 \times N_2$, $Q_1 \subset N_1$ et $Q_2 \subset N_2$ :

$\mathrm{pre}[\mathrm{R}](Q_2) = \{\mathrm{q} \mid \mathrm{q} \in N_1 \wedge \exists \mathrm{q'}.(\mathrm{q'} \in Q_2\ (\mathrm{q},\mathrm{q'}) \in \mathrm{R})\}$
$\mathrm{post}[\mathrm{R}](Q_1) = \{\mathrm{q'} \mid \mathrm{q'} \in N_2 \wedge \exists \mathrm{q}.(\mathrm{q} \in Q_1 \wedge (\mathrm{q'},\mathrm{q}) \in \mathrm{R})\ \}$
$\widetilde{pre}[\mathrm{R}](Q_2) = \overline{pre[R](\overline{Q_2})}$
$\widetilde{post}[\mathrm{R}](Q_1) = \overline{post[R](\overline{Q_1})}$

Si R est totale, alors $\widetilde{pre}[\mathrm{R}] \subset \mathrm{pre}[\mathrm{R}]$ et si R est une fonction, alors $\widetilde{pre}[\mathrm{R}] = \mathrm{pre}[\mathrm{R}]$. Pour les systèmes de transitions, les labels peuvent être considérés comme des références sur la relation qu'il labellise. La définition suivante est utilisée où $\mathcal{R} \subset (\mathrm{N} \times \mathrm{L} \times \mathrm{N})$ et $\mathrm{Q} \subset \mathrm{N}$ :

$$\mathrm{pre}[\mathrm{l}](\mathrm{Q}) = \{\ \mathrm{q} \mid \mathrm{q} \in \mathrm{N} \wedge \exists\ \mathrm{q'}.(\mathrm{q'} \in \mathrm{Q} \wedge (\mathrm{q} \xrightarrow{l} \mathrm{q'}) \in \mathcal{R})\}$$
$$\mathrm{post}[\mathrm{l}](\mathrm{Q}) = \{\ \mathrm{q'} \mid \mathrm{q'} \in \mathrm{N} \wedge \exists\ \mathrm{q}.(\mathrm{q} \in \mathrm{Q} \wedge (\mathrm{q} \xrightarrow{l} \mathrm{q'}) \in \mathcal{R})\}$$

### 5.6.2 Principe de construction d'un système de transitions fini

Dans cette section, nous étudierons une manière de construire un système de transitions étiquetées [16], défini dans le paragraphe précédent (5.6.1), depuis une spécification d'un système d'événements B.

### 5.6.3 Enumération des états

La manière la plus simple de définir un système de transitions étiquetées depuis une machine abstraite B est d'énumérer les états et les transitions. On utilise un système abstrait S avec une liste de variables $X = \{\ x_1, \ldots, x_n\}$, un invariant $I$, une initialisation $U$, une liste d'événement $(\varepsilon_i = E_i)$ pour $\mathrm{i} \in 1..\mathrm{m}$, où les états initiaux sont les affectations $\sigma_0$ :

$\sigma_0 = \{\ x_j \mapsto v_j \mid <\mathrm{U}>\ (x_j = v_j)\ \}\ \forall\ \mathrm{j} \in 1..\mathrm{n}$
où $<\mathrm{U}>(x_j = v_j)$ est le conjugué $\neg[\mathrm{U}]\neg\mathrm{P}$ de $[\mathrm{U}]\mathrm{P}$ qui dénote la plus faible précondition (cf 2.3.5) pour que la substitution d'initialisation U se termine et établisse P ($\mathrm{P} \mathrel{\widehat{=}} (x_j = v_j)$).

Pour chaque état $\sigma_p$ qui satisfait l'invariant $I$ et chaque événement $\varepsilon_i$, tel que cet événement est activé à l'état $\sigma_p$, les successeurs de $\sigma_p$ par la transition labellisée $\varepsilon_i$ sont des états et les valeurs obtenues après la substitution généralisée $E_i$ :

$\sigma_{p+1} = \{\ x_j \mapsto v'_j \mid [\sigma_p]\ (<E_i>\ (x_j = v'_j))\}\ \forall\ \mathrm{j} \in 1..\mathrm{n}$

Dans le B-book [3] (page 296), ceci signifie, si $(x_j \mapsto v_j) \in \sigma_p$ alors :

$(v_j, v'_j) \in rel_{x_j}(E_i)$

Dans cette notation $rel_{x_j}$ est une relation binaire qui donne la valeur de $x_j$ avant et après la substitution $E_i$. A cause de nos suppositions à propos de la terminaison (étant donnée l'hypothèse sur les événements, qui est que la partie action $A$ de l'événement "termine" au sens B, signifie que le prédicat term(A) est vrai), le lien entre l'événement B $\varepsilon_i = E_i$ et les transitions associées au système de transitions est :

$\mathrm{post}[\varepsilon_i] = \mathrm{rel}(E_i)$

On en déduit $\mathrm{pre}[\varepsilon_i] = \mathrm{rel}(E_i)^{-1}$ et $\widetilde{pre}[\varepsilon_i] = \mathrm{str}(E_i)$. Nous sommes intéressés par les états atteignables : ces états peuvent être construit inductivement depuis l'état initial.

## 5.7 Conclusion

L'intégration des deux types de méthodes, formelle et semi-formelles, mettent en jeu essentiellement deux diagrammes qui sont : les diagrammes de classes et d'états-transitions. Ceci permet une modélisation abstraite des structures de données et du comportement du système spécifié à la fois formellement et à l'aide de vues graphiques. La dérivation des diagrammes de classes en B fait apparaître des structures de données qui ne sont pas forcément naturelles tandis que la démarche utilisée lors de la dérivation des diagrammes d'états-transitions semble uniforme ou presque quelque soit les travaux considérés. Certains travaux utilisent d'autres diagrammes comme les diagrammes de cas d'utilisation et d'activités. Ces travaux sont très intéressants, mais il s'est avéré que la dérivation des modèles UML engendre des machines B compliquées et difficiles à comprendre d'une part et d'autre part leur dérivation est délicate à mettre en oeuvre sur certains point comme le montre les sections 5.4.2 et 5.3. D'autre part, il est difficile de projeter les obligations de preuves non résolues dans la machine B, lors de la vérification formelle, sur les vues UML correspondantes. Pour apporter un autre point de vue sur la dérivation et pour permettre d'écrire directement du B tout en obtenant des vues UML de la spécification nous travaillerons sur l'extraction de diagrammes UML. Comme la totalité des travaux déjà réalisés dans le domaine de l'intégration des méthodes formelles et semi-formelles et en particulier dans celui de la dérivation, notre démarche s'appuiera sur l'extraction des diagrammes de classes et d'états-transitions ainsi que des diagrammes plus concrets comme ceux de séquence et d'activités.

# Chapitre 6

# Extraction de diagrammes de classes

## 6.1 Introduction

Dans ce chapitre, nous présentons notre approche de transformation des machines B en diagramme de classes. Nous traitons à travers cette extraction uniquement les prédicats de typage de l'invariant. Plus précisément nous nous focalisons sur l'obtention d'une traduction à partir des ensembles (abstraits ou énumérés) définis dans une machine et des données (constantes et variables) typées à partir de ces ensembles. Les autres prédicats de l'invariant ainsi que les pré et post-conditions des opérations seront traités dans le chapitre 7. Cependant ces deux chapitres ne traitent pas les expressions d'arbres et d'enregistrement (records) car elles sont peu utilisées et ne possèdent pas une traduction directe à l'aide d'une expression équivalente en UML. Ce chapitre qui est une extension des papiers [98, 96] est divisé en quatre parties. Dans la première nous décrivons notre approche sur une machine abstraite, dans la deuxième nous montrons sur un exemple simple les résultats obtenus, dans la troisième nous étendons notre approche pour traiter la composition de plusieurs machines B et dans la quatrième nous comparons notre approche à d'autres propositions [7, 32] faites plus récemment.

## 6.2 De la spécification B au diagramme de classes

### 6.2.1 Les machines abstraites

Une spécification B étant structurée en machines abstraites, il est nécessaire de trouver une équivalence en UML pour cette notion de machine. Dans les techniques de dérivation d'UML vers B, la solution choisie consiste à représenter une classe par une machine abstraite. C'est une solution naturelle dans la mesure où la classe en UML et la machine abstraite en B sont les éléments de granularité élémentaire qui permettent de regrouper des données et des opérations. C'est pourquoi nous choisissons de représenter une machine abstraite par une classe comme dans l'exemple 6.1 et la figure 6.2.

### 6.2.2 Les variables entières et booléennes

Ces variables définies dans la machine (Fig 6.3) deviennent des attributs de la classe (Fig 6.4) qui représente cette machine. Dans la machine de l'exemple présenté dans la figure

MACHINE
*AM*
...
OPERATIONS
$Op1 \widehat{=} ...;$
$Op2 \widehat{=} ...$
END

FIG. 6.1 – La machine *AM*

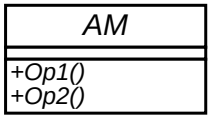


FIG. 6.2 – Modélisation de la machine *AM* en classe

6.3, $b$ est une variable booléenne et $x$ est un entier naturel. L'initialisation des variables est obligatoire. Le typage des paramètres et des résultats d'une opération B se fait dans le corps de celle-ci au niveau des substitutions.

MACHINE
*AM*
VARIABLES
$b, x$
INVARIANT
$b \in BOOL \wedge x \in NAT$
INITIALISATION
$b := true \parallel x := 0$
OPERATIONS
$Op1(y) \widehat{=}$PRE $y \in NAT$ THEN ... END;
$z \leftarrow Op2 \widehat{=}$PRE ... THEN $z := x$ END
END

FIG. 6.3 – Machine munie de variables entières et booléennes

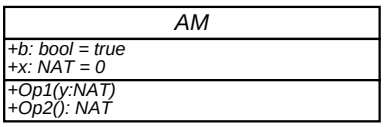


FIG. 6.4 – Modélisation des variables

### 6.2.3 Les ensembles simples

#### 6.2.3.1 Les ensembles abstraits et les variables associées

Un ensemble abstrait (comme $A$ dans l'exemple présenté figure 6.5) est représenté d'une part par une classe (notée ici *E_A*) correspondant au type d'éléments de l'ensemble et d'autre part par une association entre cette classe et la classe représentant la machine (fig 6.6). L'extrémité de l'association du côté de la classe correspondant au type d'élément possède un nom (*rôle*) qui est celui de l'ensemble (ici $A$). Les variables de type ensemble deviennent des associations avec leur nom comme nom de rôle du côté correspondant au type d'élément. Une variable ($a$) appartenant à un ensemble abstrait ($A$) est représentée par une agrégation entre la classe représentant la machine et la classe correspondant au type d'élément avec comme nom de rôle du côté de cette classe le nom de la variable.

MACHINE
    $AM$
SETS
    $A$
VARIABLES
    $B, a$
INVARIANT
    $B \subseteq A \wedge a \in A$
...
END

FIG. 6.5 – Machine munie d'ensembles abstraits

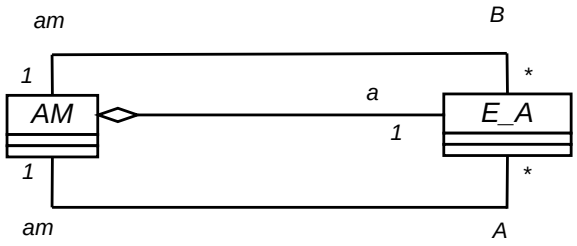


FIG. 6.6 – Modélisation des variables typées avec des ensembles abstraits

Une contrainte OCL permet d'exprimer l'inclusion entre les ensembles $A$ et $B$ qui n'apparaît pas sur le diagramme de classes ainsi que l'appartenance de $a$ à l'ensemble $A$. Dans la contrainte de la figure 6.7, le contexte par rapport auquel s'exprime la contrainte est la classe *AM* et **inv** signifie invariant. En utilisant le rôle *A* depuis cette classe, on obtient un ensemble. La fonction prédéfinie **includesAll** permet de s'assurer que tous les éléments de l'ensemble passé en paramètre (obtenu par le rôle *B*) sont inclus dans l'ensemble obtenu par le rôle *A*.

**context** *AM* **inv :**
*A*→**includesAll(***B***)**
**and** *A*→**includes(***a***)**

FIG. 6.7 – Contraintes d'inclusion dans un ensemble $A$

#### 6.2.3.2 Les ensembles énumérés

Un ensemble énuméré (comme C dans l'exemple de la figure 6.8) est représenté de façon similaire à un ensemble abstrait, mais ici le cardinal de l'ensemble est connu, ainsi que les différentes valeurs possibles de ses éléments. La classe représentant les éléments de l'ensemble *E_C* contient un attribut *value* dont le type est défini par une énumération UML reflétant les différentes valeurs possibles.

MACHINE
*AM*
SETS
$C = \{c1, c2, c3\}$
VARIABLES
$D, c$
INVARIANT
$D \subseteq C \wedge c \in C$
$\wedge\ c = c1$
...
END

FIG. 6.8 – Machine munie d'ensembles énumérés

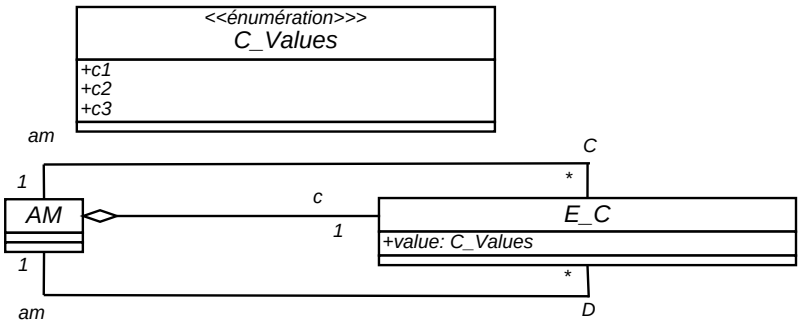


FIG. 6.9 – Modélisation des ensembles énumérés et variables typées avec ceux-ci

De même que pour les ensembles abstraits, une contrainte permet d'exprimer ce qui n'apparaît pas sur le diagramme de classes.

**context** *AM* **inv :**
*C*→**includesAll(***D***)**
**and** *C*→**includes(***c***)**
**and** *c.value*=*C_Values*::*c1*

FIG. 6.10 – Contrainte relative à l'inclusion, l'appartenance et l'égalité

### 6.2.4 Les ensembles de couples

#### 6.2.4.1 Le produit cartésien

Le produit cartésien de deux ensembles $X$ et $Y$, noté $X \times Y$, est l'ensemble dont les éléments sont tous les couples (les paires ordonnées) dont le premier élément appartient à $X$ et le deuxième élément à $Y$. Le produit cartésien de $X \times Y$ est représenté par une classe *E_X_Y* et par une association entre cette classe et la classe représentant la machine. Le rôle du côté de la classe nommée *X_Y* représente le produit cartésien. Dans l'exemple de la figure 6.11, l'ensemble $A$ est inclus dans le produit cartésien $X$ par $Y$. $A$ est donc un ensemble de couples.

MACHINE
*AM*
SETS
*X ; Y*
VARIABLES
*A*
INVARIANT
$A \subset X \times Y$
...
END

FIG. 6.11 – Machine munie de produit cartésien

Remarque : Le produit cartésien n'est pas forcément la notation la plus usitée en B pour définir des ensembles de couples mais elle nous permet d'obtenir dans la traduction, le typage

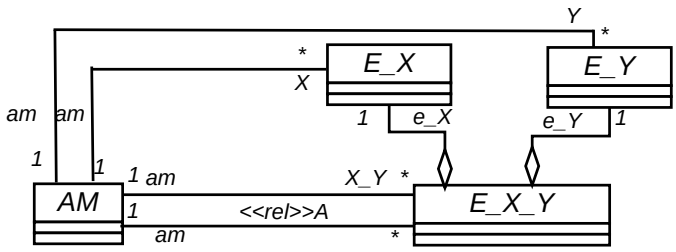


FIG. 6.12 – Modélisation du produit cartésien

d'un couple par la définition de la classe représentant les éléments du produit cartésien. Dans l'exemple de la figure 6.11, tous les ensembles de couples ayant comme premier élément un élément de $X$ et comme deuxième élément un élément de $Y$ seront définis comme des rôles reliant la machine à la classe $E\_X\_Y$.

#### 6.2.4.2 Les relations

En B, les relations sont des ensembles de couples $xi \mapsto yi$ avec $xi$ appartenant à l'ensemble $X$ et $yi$ appartenant à l'ensemble $Y$. Une relation $R$ peut être définie dans l'invariant par la notation $R : X \leftrightarrow Y$ qui signifie que $R$ appartient à l'ensemble des relations de l'ensemble $X$ dans l'ensemble $Y$. Deux cas peuvent se présenter : le premier si $X$ et $Y$ sont des ensembles simples, le deuxième si $X$ et/ou $Y$ est une relation. Les deux cas sont illustrés sur l'exemple de la figure 6.13 et de la figure 6.14, le premier par $R1$ et le deuxième par $R2$.

MACHINE
    *AM*
SETS
    $X ; Y$
VARIABLES
    *R1, R2*
INVARIANT
    $R1 \in X \leftrightarrow Y \wedge$
    $R2 \in R1 \leftrightarrow Y$
...
END

FIG. 6.13 – Machine utilisant des relations

Remarque : La notation exprimée sur la figure 6.14 n'est pas la plus concise possible. Si dans la spécification B, un type de couple n'est trouvé qu'une fois alors la traduction peut être rendue plus lisible en utilisant comme nom de la classe le nom de la relation au lieu du nom du produit cartésien. La figure 6.15 est une autre traduction de la spécification B de la figure 6.13 qui suit ce principe. Le domaine et le codomaine pour les ensembles de couples (produit

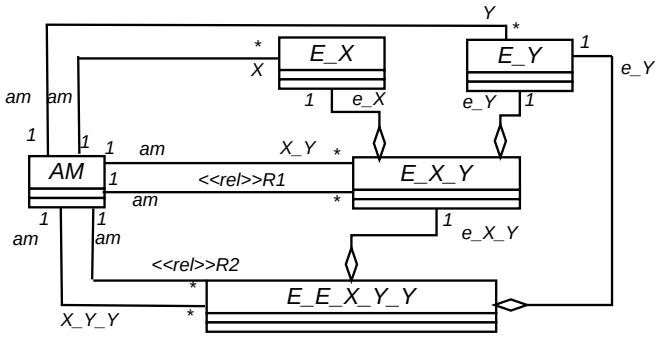


FIG. 6.14 – Modélisation des relations

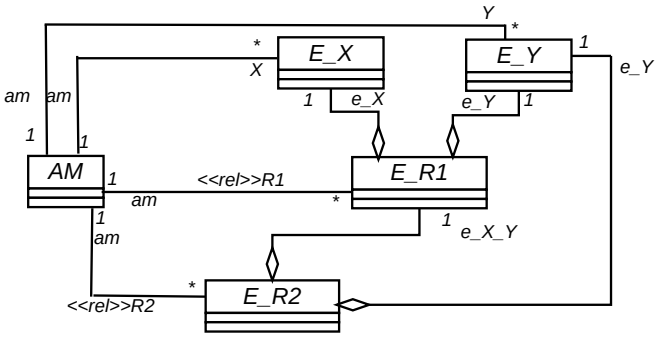


FIG. 6.15 – Une autre modélisation des relations

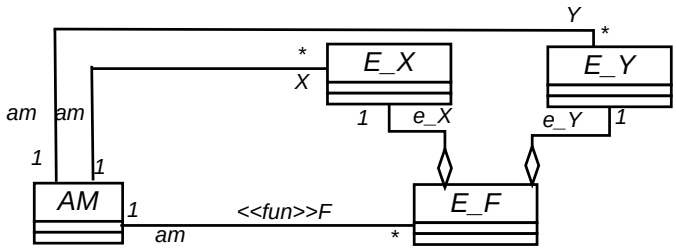


FIG. 6.16 – Diagramme de classes d'une machine avec une fonction $F$ de $X$ dans $Y$

cartésien, relation, fonction, injection, bijection et surjection) sont définis sur le diagramme de classes par des noms de rôles.

Les objets instanciés des classes *E_R1* et *E_R2* représentent chacun un couple des relations B respectives *R1* et *R2*.

### 6.2.4.3 Les fonctions

Les fonctions sont des relations qui ne contiennent pas deux couples distincts ayant le même premier élément. Une fonction se traduit exactement comme une relation (figure 6.14). Afin de caractériser les différents types de fonction (injection, surjection, bijection) partielle ou totale, des stéréotypes (paragraphe 6.2.5) sont ajoutés sur le diagramme de classe ainsi que des contraintes OCL. Si le F de la figure 6.16 représente une fonction totale (stéréotype <<fun>>) de la spécification B alors la contrainte OCL, présenté figure 6.17, est ajoutée.

La contrainte OCL relative aux fonctions totales (fig 6.17) signifie que pour deux objets différents de la classe *E_F* (cette classe représente la fonction totale), les objets correspondants

**context** *AM* **inv :**
*F*→**forAll**(*c1, c2 : E_F* | *c1 <> c2* **implies** *c1.e_X <> c2.e_X*)
**and** *F*→**size()**= *X*→**size()**

FIG. 6.17 – Contrainte relative aux fonctions totales

au rôle *e_X* sont différents et le nombre d'objets de la classe *E_F* est le même que celui de la classe *E_X*.

Les fonctions partielles se traitent de la même manière que les fonctions totales, à l'exception de la contrainte sur le nombre d'éléments de l'ensemble $F$ qui doit être inférieur ou égal au nombre d'éléments de l'ensemble de départ $X$. Pour une fonction $F$ partielle, la contrainte précédente (fig 6.17) s'écrit comme dans la figure 6.18. Cette différence apparaît aussi entre les injections, bijections et surjections totales et partielles.

**context** *AM* **inv :**
*F*→**forAll (***c1, c2 : E_F | c1 <> c2* **implies** *c1.e_X <> c2.e_X*
**and** *F*→**size() ≤** *X*→**size()**

FIG. 6.18 – Contrainte relative aux fonctions partielles

#### 6.2.4.4 Injection

L'expression d'une fonction d'injection totale revient à donner les contraintes d'une fonction totale, plus une contrainte supplémentaire qui exprimera l'injection comme dans la figure 6.19.

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F | c1 <> c2* **implies** *c1.e_X <> c2.e_X***)**
**and** *F* → **forAll (***c1, c2 : E_F | c1.e_X <> c2.e_X* **implies**
*c1.e_Y <> c2.e_Y***)**
**and** *F* → **size()=** *X* → **size()**

FIG. 6.19 – Contrainte relative aux injections totales

La contrainte OCL relative aux injections totales (fig 6.17) signifie que pour deux objets différents de la classe *E_F* (cette classe représente l'injection totale), les objets correspondant au rôle *e_X* sont différents. Dans ce cas les objets correspondant au rôle *e_Y* le sont aussi. Le nombre d'objets de la classe *E_F* est le même que celui de la classe *E_X*.

Pour une injection $F$ partielle, la contrainte s'écrit comme dans la figure 6.20.

**context** *AM* **inv :**
*F*→ **forAll(***c1,c2 : E_F | c1 <> c2* **implies***c1.e_X <> c2.e_X***)**
**and** *F*→ **forAll (***c1, c2 : E_F | c1.e_X <> c2.e_X***implies**
*c1.e_Y <> c2.e_Y***)**
**and** *F* → **size()≤** *X* → **size()**

FIG. 6.20 – Contrainte relative aux injections partielles

#### 6.2.4.5 Surjection

Pour une fonction de surjection totale, elle s'exprime de la même manière qu'une injection totale à l'exception de la contrainte relative à la surjection (fig 6.21).

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F | c1 <> c2* **implies** *c1.e_X <> c2.e_X***)**
**and** *F* → **collect (***c : E_F | c.e_Y***)** → **includesAll(***Y***)**
**and***F* → **size()**= *X*→ **size()**

FIG. 6.21 – Contrainte relative aux surjections totales

La contrainte OCL relative aux injections totales (fig 6.21) signifie que pour deux objets différents de la classe *E_F* (cette classe représente la surjection totale), les objets correspondant au rôle *e_X* sont différents. Chaque objet de la classe *E_Y* correspond à une occurrence du rôle *e_Y* d'un objet de la classe *E_F*. Le nombre d'objets de la classe *E_F* est le même que celui de la classe *E_X*.

Pour une surjection *F* partielle, la contrainte s'écrit comme dans la figure 6.22.

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F | c1 <> c2* **implies** *c1.e_X <> c2.e_X***)**
**and** *F* → **collect (***c : E_F | c.e_Y***)** → **includesAll(***Y***)**
**and** *F* → **size()** ≤ *X* → **size()**

FIG. 6.22 – Contrainte relative aux surjections partielles

#### 6.2.4.6 Bijection

Une fonction de bijection totale désigne en même temps une fonction d'injection et de surjection totale. Son expression en OCL est donnée en combinant les contraintes d'une fonction d'injection et de surjection totale (fig 6.23 ou 6.24).

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F | c1 <> c2* **implies** *c1.e_X <> c2.e_X***)**
**and** *F*→ **forAll (***c1, c2 : E_F | c1.e_X <> c2.e_X* **implies**
*c1.e_Y <> c2.e_Y***)**
**and** *F* → **collect (***c : E_F | c.e_Y***)** → **includesAll(***Y***)**
**and** *F*→ **size()**= *X* → **size()**

FIG. 6.23 – Contrainte relative aux bijections totales

Pour une bijection *F* partielle, la contrainte s'écrit comme dans la figure 6.25.

### 6.2.5 Utilisation de stéréotypes

L'extraction de produits cartésiens, relations, fonctions, injections, bijections et surjections partielles ou totales s'exprime de la même façon sur le diagramme de classes. D'un point de vue

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F* | *c1* <> *c2* **implies** *c1.e_X* <> *c2.e_X*
**and** *c1.e_Y* <> *c2.e_Y***)**
**and** *F*→ **size()**= *X*→ **size() and** *F* → **size()**= *Y*→ **size()**

FIG. 6.24 – Autre écriture possible de la contrainte relative aux bijections totales

**context** *AM* **inv :**
*F* → **forAll(***c1,c2 : E_F* | *c1* <> *c2* **implies** *c1.e_X* <> *c2.e_X***)**
**and** *F* → **forAll (***c1, c2 : E_F* | *c1.e_X* <> *c2.e_X* **implies**
*c1.e_Y* <> *c2.e_Y***)**
**and** *F*→ **collect (***c : E_F* | *c.e_Y***)** → **includesAll(***Y***)**
**and** *F* → **size()** ≤ *X* → **size()**

FIG. 6.25 – Contrainte relative aux bijections partielles

graphique on ne peut pas voir la différence entre toutes ces entités. Afin de mieux caractériser les données extraites de la spécification et de mieux établir le lien avec ces dernières nous utilisons des stéréotypes qui seront visibles sur le diagramme de classes permettant ainsi d'identifier visuellement les différents types d'ensembles de couples. La liste est définie dans le tableau 6.1.

| Nom | Représentation en B | Stéréotype |
|---|---|---|
| Relation | $\leftrightarrow$ | <<*rel*>> |
| Fonction totale | $\rightarrow$ | <<*fun*>> |
| Fonction partielle | $\nrightarrow$ | <<*pfun*>> |
| Injection totale | $\rightarrowtail$ | <<*inj*>> |
| Injection partielle | $\nrightarrowtail$ | <<*pinj*>> |
| Surjection totale | $\twoheadrightarrow$ | <<*surj*>> |
| Surjection partielle | $\nrightarrow\!\!\!\!\twoheadrightarrow$ | <<*psurj*>> |
| Bijection totale | $\rightarrowtail\!\!\!\!\twoheadrightarrow$ | <<*bij*>> |
| Bijection partielle | $\nrightarrowtail\!\!\!\!\twoheadrightarrow$ | <<*pbij*>> |

TAB. 6.1 – Stéréotypes d'aide à la lecture des diagrammes de classes

## 6.3 Un exemple simple

### 6.3.1 Etude de cas : Inscription d'étudiants à un diplôme

L'objectif de cet exemple simplifié est la gestion de l'inscription d'étudiants à un diplôme. L'inscription ne suffit pas et il faut que les étudiants s'inscrivent aussi aux examens. La spécification B de l'exemple est donnée figure 6.26.

MACHINE
*GestionDesInscriptions*
SETS
*ETUDIANTS ; DIPLOMES*
VARIABLES
*Inscrits, InscritsAuxExamens*
INVARIANT
$Inscrits \in ETUDIANTS \nrightarrow DIPLOMES \wedge$
$InscritsAuxExamens \subset Inscrits$
INITIALISATION
$Inscrits := \emptyset \parallel InscritsAuxExamens := \emptyset$
OPERATIONS
*inscrire(etudiant, diplome)* $\widehat{=}$
PRE
$etudiant \in ETUDIANTS \wedge diplome \in DIPLOMES \wedge$
$etudiant \notin dom(Inscrits)$
THEN
$Inscrits := Inscrits \cup \{etudiant \mapsto diplome\}$
END ;
....
END

FIG. 6.26 – Exemple de la gestion des inscriptions

### 6.3.2 Modélisation UML correspondante

Les ensembles (clause SETS) *ETUDIANTS* et *DIPLOMES* permettent de représenter tous les étudiants et tous les diplômes potentiels. Les variables (clause VARIABLES) *Inscrits* et *InscritsAuxExamens* sont typées dans l'invariant (clause INVARIANT). *Inscrire* est une fonction partielle entre l'ensemble des étudiants et celui des diplômes : c'est donc un ensemble de couples où le premier élément d'un couple est un étudiant et le deuxième un diplôme. *InscritsAuxExamens* est un sous ensemble de *Inscrits*.

Initialement aucun étudiant n'est inscrit. L'invariant est devenu vrai après l'initialisation (clause INITIALISATION). Dans la clause OPERATIONS on trouve l'ensemble des opérations. La seule opération décrite ici est *Inscrire* qui correspond à l'inscription d'un étudiant en vue d'obtenir un diplôme. Cette opération est soumise à une précondition qui doit être vérifiée. Cette précondition permet d'attribuer un type aux paramètres puis de vérifier que l'étudiant demandant à inscrire n'est pas déjà inscrit. Le corps de l'opération permet d'ajouter le couple *etudiant*$\mapsto$*diplome* à l'ensemble *Inscrits*.

Sur le diagramme de la figure 6.27 une partie des règles adoptées apparaît :
- La machine est une classe.
- Un ensemble abstrait comme *ETUDIANTS* est représenté par une classe *E_ETUDIANTS* correspondant au type d'éléments de l'ensemble et par une association entre cette classe et la classe représentant la machine. L'extrémité de l'association du côté de la classe correspondant au type d'éléments possède un nom (rôle) qui est celui de l'ensemble

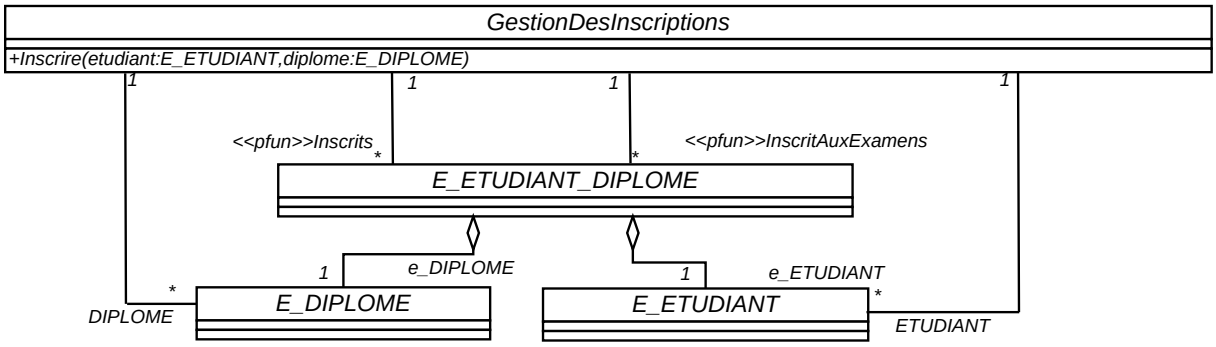


FIG. 6.27 – Diagramme de classes correspondant à la machine abstraite *GestionDesInscriptions*

**context** *GestionDesInscriptions* **inv :**
*Inscrits*→**includesAll(***InscritsAuxExamens***)**

FIG. 6.28 – Contrainte d'inclusion ensembliste de *InscritsAuxExamens* dans *Inscrits*

(ici *ETUDIANTS*). Les variables de type ensemble deviennent des associations avec leur nom comme nom de rôle du côté correspondant au type d'éléments.

- La fonction partielle *Inscrits* est traduite par une classe *E_ETUDIANT_DIPLOME* reliée par une association dont le rôle est *Enrolled* du côté de la classe représentant la machine. Cette classe est également reliée par une agrégation à la classe *E_ETUDIANT* et par une autre à la classe *E_DIPLOME*.
- Le processus est le même pour la fonction partielle *InscritsAuxExamens*. Pour indiquer que *InscritsAuxExamens* est un sous ensemble de *Inscrits*, on utilise la contrainte OCL de la figure 6.28.

En fait ceci ne suffit pas pour décrire une fonction partielle comme *Inscrits* :

- Une fonction est un ensemble de couples qui ne contient pas deux couples distincts ayant le même premier élément.
- Une fonction est partielle lorsque tous les éléments de l'ensemble de départ (*ETUDIANTS*) ne sont pas utilisés pour faire des couples.

Il faut donc rajouter la contrainte OCL 6.29 qui permet de restreindre l'association au cas des fonctions partielles.

**context** *GestionDesIncriptions* **inv :**
*Inscrits*→**forAll(***c1, c2 : E_ETUDIANT_DIPLOME | c1 <> c2*
**implies** *c1.e_ETUDIANT <> c2.e_ETUDIANT***)**
**and** *Inscrits*→**size()**≤ *ETUDIANT*→**size()**

FIG. 6.29 – Contrainte relative à la fonction partielle *Inscrits*

## 6.4 Mécanismes de composition

Les relations de visibilité et d'instanciation entre machines peuvent être montrées à l'intérieur du diagramme de classes.

Les visibilités entre UML et B sont différentes. En B il existe une visibilité dite *visible non modifiable* pour les variables et opérations (les variables ne sont pas modifiables donc les opérations ne modifient pas les variables de machine cf 2.3.8) qui n'existe pas en UML. D'autre part la visibilité en B est permise par l'utilisation de mécanisme de composition au contraire d'UML où la visibilité est définie à la déclaration de l'attribut et éventuellement par l'appartenance ou non au package. Pour toutes ces raisons il n'est pas possible de réaliser une traduction complète et exacte de la visibilité définie dans les spécifications. Néanmoins la preuve permet de bloquer les accès non autorisés aux variables.

### 6.4.1 Visibilité à travers les clauses SEES et IMPORTS

Les clauses SEES et IMPORTS permettent de voir variables et opérations. La clause SEES permet une visibilité *visible non modifiable* alors que la clause IMPORTS permet la modification par les opérations (opérations qui modifient la valeur des variables) et par les variables. Par conséquent sachant que la preuve bloque les accès non autorisés, on peut imaginer une visibilité identique pour les attributs des machines citées dans les deux clauses une fois la machine traduite en classe UML. Cette visibilité sera définie comme *public* pour ne pas être plus restrictive que celle de B sachant que la preuve a bloqué les accès interdits. Il existe une différence entre les deux clauses, la clause SEES permet juste de voir les entités de la machine citée alors que la clause IMPORTS permet d'instancier la machine importée. Il est donc nécessaire de transcrire l'instanciation par une association. On pourrait aussi déclarer une variable du type de la classe utilisée mais symboliquement on ne voit pas le lien. Dans le but de mieux documenter le système il est préférable d'utiliser une agrégation pour signifier clairement l'existence du lien et de l'instance.

### 6.4.2 Clause EXTENDS, INCLUDES et PROMOTES

Les attributs et méthodes des classes sont en visibilité *public*, visibilité moins restrictive que dans les spécifications B. Toutefois le problème provient des opérations non promues qui sont visibles (visibilité *public*). En effet il n'existe pas en UML de mécanisme équivalent à la clause PROMOTES. Cette clause permet à la machine qui inclut une autre machine de promouvoir des opérations de la machine incluse. En UML soit aucune opération n'est visible soit elles le sont toutes. Le problème se circonscrit à la visibilité car la preuve bloque les accès illégaux. La représentation de la clause INCLUDES sera réalisée par un lien de type association sachant que les attributs et opérations de la machine incluse auront une visibilité *public* pour maintenir la visibilité définie en B. Pour la clause EXTENDS, le cas est différent car elle permet l'utilisation des méthodes de la machine étendue traduite en UML dans la machine qui l'étend. Il est donc nécessaire d'utiliser une généralisation pour maintenir le modèle UML juste (possibilité d'utilisation des méthodes de la classe générale dans les classes filles). D'autre part l'utilisation de la clause PROMOTES oblige l'utilisation d'une généralisation dans la traduction d'une clause INCLUDES pour les mêmes raisons.

## 6.5 Comparaison avec d'autres propositions de traduction

### 6.5.1 Introduction

Les objectifs que nous nous étions fixés étaient de permettre la visualisation à partir d'une machine abstraite B de façon automatisée et systématique des vues graphiques UML. Cette automatisation permet au spécifieur B où à la personne chargée d'évaluer la spécification B à partir du modèle UML de ne pas perdre de temps dans la recherche de la bonne visualisation. Cette automatisation peut aussi conduire à ce que cette visualisation ne soit pas la plus adaptée du point de vue de la lisibilité ou ne conduise pas aux modèles UML les plus naturels. Cette restriction est peu importante puisqu'il existe plusieurs modèles possibles pour exprimer un système répondant au même cahier des charges sans que l'on puisse affirmer avec certitude que l'une de ces solutions soit la meilleure. Néanmoins pour obtenir un modèle UML mieux adapté aux critères personnels des spécifieurs qui l'utiliseront, la proposition de [32] consiste à envisager une traduction interactive et incrémentale qui permettra à l'utilisateur de choisir à chaque étape entre plusieurs alternatives de traduction afin d'obtenir une visualisation naturelle. Nous reprenons le niveau abstrait de l'exemple proposé dans [32] en montrant les différentes solutions envisagées dans cet article, puis en comparant avec ce que nous obtiendrions par notre méthode.

### 6.5.2 Etude de cas : système de contrôle d'accès aux batiments

L'exemple considéré (spécification B présenté figure 6.30) est celui de l'étude de cas du système de contrôle d'accès aux bâtiments :

- les acteurs du système (les personnes et les bâtiments) sont représentés par les ensembles abstraits *BATS* et *PERS*,
- les autorisations d'accès des personnes aux bâtiments sont matérialisées par la relation *aut*,
- la situation des personnes dans les bâtiments est représentée par la fonction totale *sit*.

MACHINE
*ControleAccesBatiments*
SETS
*BATS ; PERS*
VARIABLES
*aut, sit*
INVARIANT
$aut \in PERS \leftrightarrow BATS \wedge$
$sit \in PERS \rightarrow BATS \wedge$
$sit \subseteq aut \wedge$
...
END

FIG. 6.30 – Extrait de la spécification B du contrôle d'accès aux bâtiments

### 6.5.3 Les traductions en UML

#### 6.5.3.1 L'approche interactive et incrémentale

Dans l'approche choisie par [32], les ensembles abstraits sont représentés par une classe car ils possèdent des relations avec d'autres entités (relation *aut* et fonction totale *sit*) autrement ils seraient représentés par un type de base. A partir de là, les auteurs envisagent trois solutions représentées sur la figure 6.31 :

- La relation *aut* décrit une relation entre deux ensembles représentés par des classes. Cette relation peut donc être modélisée par une association ayant la multiplicité (*,*) entre ces classes. Une fois ce choix effectué, il reste deux possibilités pour la fonction totale *sit* :
  - Elle est décrite par une association de multiplicité (1,*)
  - Elle devient un attribut de la classe *PERS*
- La relation *aut* peut aussi être représentée par un attribut multivalué dans la classe *PERS* pour indiquer l'ensemble des bâtiments autorisés pour une personne donnée. Dans ce cas, la fonction totale *sit* est un attribut de la classe *PERS*.

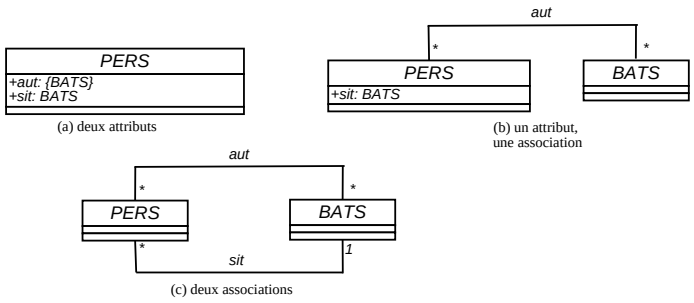


FIG. 6.31 – Les trois solutions possibles en UML

L'inclusion entre *sit* et *aut* est traduite par une contrainte OCL.

#### 6.5.3.2 Notre approche

Dans notre solution, nous avons choisi de traiter les ensembles (abstraits, énumérés, relations, fonctions, ...) de façon uniforme en donnant des rôles (nom de l'ensemble) aux associations entre la classe représentant la machine abstraite et les classes représentant les éléments des ensembles. Cette solution est moins naturelle à première vue mais ce handicap est rattrapé, de notre point de vue, par le fait qu'elle est systématique et automatisable. De plus nous montrons dans les chapitres suivant que la génération à partir de l'invariant et des opérations B de contraintes OCL est faisable et automatisable. Le grand désavantage de notre approche est la nécessité de rajouter des contraintes OCL pour préciser les différents types de fonction.

### 6.5.4 Comparaisons de ces deux approches

L'avantage essentiel de notre approche est la possibilité de traduire une machine B quelconque de façon automatique. Dans l'approche [32], la modélisation peut apparaître plus simple mais nous pensons qu'il y a plusieurs inconvénients :

- Pour l'instant rien n'est précisé au niveau de la traduction de la machine abstraite ce qui ne permet pas de définir des variables telles que $pers \in PERS$ ;

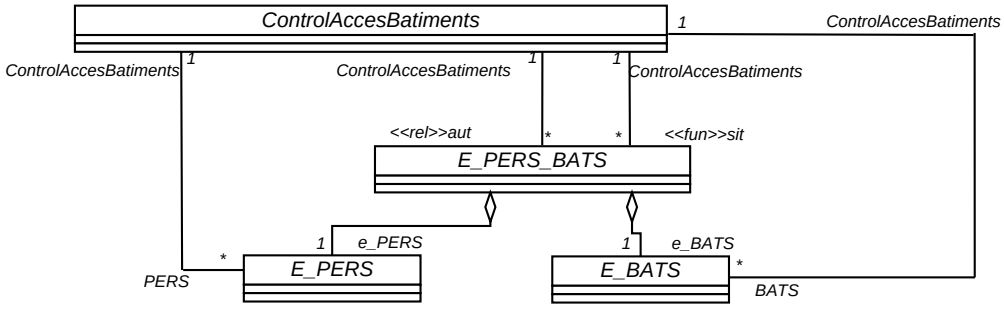


FIG. 6.32 – Notre proposition

- les ensembles ne sont pas traités de manière uniforme : tantôt ils sont traités comme des classes, tantôt comme des attributs, tantôt comme des associations. Ce peut être judicieux d'avoir une telle flexibilité à condition de prendre quelques précautions. En effet, les exemples suivants ne nous paraissent pas faciles à traduire :
  - $ens \in A \times B \rightarrow C$ où $A$, $B$ et $C$ sont des ensembles,
  - $R1 \in X \leftrightarrow Y \wedge R2 \in R1 \leftrightarrow Y$ où $X$ et $Y$ sont des ensembles,
  - la relation *aut* ne peut être réutilisée pour typer une autre variable comme dans $a1 \in aut$.
- pour finir, l'interaction avec l'utilisateur pour définir la meilleure traduction possible peut s'avérer extrêmement fastidieuse et coûteuse en temps sur des modèles B complexes car il s'agit d'une véritable réexpression de la spécification.

Un avantage que l'on peut noter dans l'approche de Fekhi, Jemni et Merz est la possibilité de visualiser les fonctions, les injections, les surjections et les bijections totales et partielles par les cardinalités d'une association directement sur le diagramme sans faire intervenir de contraintes OCL.

# Chapitre 7

# Génération de contraintes OCL

## 7.1 Introduction

Un des objectifs de nos travaux est d'étudier la mise en oeuvre des techniques qui permettent d'assister le non spécialiste de la méthode formelle B dans la compréhension de celle-ci et de favoriser la communication entre les différents acteurs du développement en permettant de visualiser les spécifications B à l'aide de diagrammes UML. Dans ce chapitre, nous allons au-delà de cet objectif en explorant comment transformer les invariants et les opérations B en contraintes OCL en se basant sur la traduction que nous avons proposée en diagrammes de classes pour les machines abstraites B. Cet aspect peut-être intéressant dans les cas suivants :

- Permettre depuis une spécification B de générer une modélisation UML. Il pourrait s'agir d'une partie critique d'un système spécifié en B [1] sur laquelle sont effectuées des preuves. La modélisation UML obtenue à partir de B complète la modélisation UML des parties non critiques du système.
- Permettre une comparaison entre l'expressivité et la lisibilité d'une spécification B et celle d'une modélisation UML/OCL. La question autour d'OCL étant de savoir si cette notation formelle bien adaptée à une notation de modélisation à objets peut exprimer de manière élégante, concise et lisible des prédicats éventuellement complexes comme le fait B.

Le travail présenté dans les sections suivantes s'est déroulé à partir de l'étude d'exemples de machines abstraites B que l'on a traduit en OCL. A partir de ces traductions, un ensemble de règles intuitives a été élaboré et les traductions B vers OCL éventuellement retravaillées en fonction des règles adoptées.

## 7.2 Réécriture en OCL des invariants B

### 7.2.1 Introduction

Les règles sont présentées dans le paragraphe suivant sous la forme de cinq tableaux d'équivalence B-OCL : les prédicats, les expressions arithmétiques, les expressions ensemblistes, les relations et les séquences. Les équivalents ont été déduits à partir des exemples de traduction intuitive en OCL de spécifications écrites en B. Certaines expressions B ont été omises comme les arbres et records (le domaine de la traduction est détaillé dans 7.2.4.2), d'autres comme la règle 5 du tableau 7.3 qui décrit l'union d'un ensemble et d'un élément (ens

$\cup$ {elem}) et la règle 6 du tableau 7.3 qui décrit la soustraction d'un élément d'un ensemble (ens - {elem}) sont des cas particuliers mais qui se présentent suffisamment fréquemment en B pour que l'on ait choisi de les traiter par une règle spécifique. Nous allons présenter ces transformations sur un exemple complet.

### 7.2.2 Tableaux d'équivalence B vers OCL

Le tableau 7.1 présente les règles de traduction des expressions de structuration des prédicats, les opérateurs ensemblistes d'appartenance, d'inclusion et d'égalité ainsi que les opérateurs arithmétiques d'égalité et d'inégalité. Ces expressions retournent un booléen {*vrai, faux*} lors de leur interprétation.

| N° | Désignation | Prédicat B | Contrainte OCL |
|---|---|---|---|
| 1 | Conjonction | *exp1* $\wedge$ *exp2* | *exp1* **and** *exp2* |
| 2 | Disjonction | *exp1* $\vee$ *exp2* | *exp1* **or** *exp2* |
| 3 | Négation | $\neg$ *exp* | *not exp* |
| 4 | Implication | *exp1* $\Rightarrow$ *exp2* | *exp1* **implies** *exp2* |
| 5 | Equivalence | *exp1* $\Leftrightarrow$ *exp2* | *exp1* **implies** *exp2* **and** *exp2* **implies** *exp1* |
| 6 | Quantificateur Universel | $\forall$*(elem).(elem* $\in$ *ens* $\Rightarrow$ *pred)* | *ens*$\rightarrow$**forAll(***elem :Type* \| *pred***)** |
| 7 | Quantificateur Existentiel | $\exists$*(elem).(elem*$\in$*ens* $\wedge$ *pred)* | *ens*$\rightarrow$**exists(***elem :Type* \| *pred***)** |
| 8 | Egalité | *exp1* = *exp2* | *exp1* = *exp2* |
| 9 | Inégalité | *exp1* $\neq$ *exp2* | *not (exp1 = exp2)* |
| 10 | Appartenance | *elem* $\in$ *ens* | *ens*$\rightarrow$ **includes (***elem***)** |
| 11 | Non appartenance | *elem* $\notin$ *ens* | *ens*$\rightarrow$ **excludes(***elem***)** |
| 12 | Inclusion | *ens1* $\subseteq$ *ens2* | *ens2*$\rightarrow$ **includesAll(***ens1***)** |
| 13 | Inclusion stricte | *ens1* $\subset$ *ens2* | *ens2*$\rightarrow$**includesAll(***ens1***) and not(***ens1=ens2***)** |
| 14 | Non inclusion | *ens1* $\not\subseteq$ *ens2* | **not(***ens2*$\rightarrow$**includesAll(***ens1***))** |
| 15 | Non inclusion stricte | *ens1* $\not\subset$ *ens2* | **not(***ens2*$\rightarrow$**includesAll(***ens1***)) or** *ens1=ens2* |
| 16 | Supérieur ou égal | *exp1* $\geq$ *exp2* | *exp1* $\geq$ *exp2* |
| 17 | Inférieur ou égal | *exp1* $\leq$ *exp2* | *exp1* $\leq$ *exp2* |
| 18 | Strictement supérieur | *exp1* $>$ *exp2* | *exp1* $>$ *exp2* |
| 19 | Strictement inférieur | *exp1* $<$ *exp2* | *exp1* $<$ *exp2* |
| 20 | Ensemble vide | *ens* = $\emptyset$ | *ens*$\rightarrow$ **isEmpty()** |
| 21 | Différent ensemble vide | *ens* $\neq$ $\emptyset$ | *ens*$\rightarrow$ **notEmpty()** |

TAB. 7.1 – Equivalence en OCL des prédicats B

Le tableau 7.2 présente les règles de traduction des expressions arithmétiques et ensem-

blistes retournant un entier pour résultat.

| N° | Désignation | Expression B | Contrainte OCL |
|---|---|---|---|
| 1 | Addition | $x + y$ | *x + y* |
| 2 | Différence | $x - y$ | *x - y* |
| 3 | Produit | $x * y$ | *x * y* |
| 4 | Division entière | $x / y$ | *x*.**div(***y***)** |
| 5 | Modulo | $x \bmod y$ | *x*.**mod(***y***)** |
| 6 | Minimum | min(*ens*) | *ens*→**iterate(***elem ; res :Integer=MAXINT* \| **if** *res*.**max(***elem***) then** *elem* **else** *res***)** |
| 7 | Maximum | max(*ens*) | *ens*→**iterate(***elem ; res :Integer=MININT* \| **if** *res*.**min(***elem***) then** *elem* **else** *res***)** |
| 8 | Cardinalité | card(*ens*) | *ens*→**size()** |

TAB. 7.2 – Equivalence en OCL des expressions arithmétiques B

Le tableau 7.3 présente les règles de traduction des opérateurs ensemblistes telles que la définition, l'union, l'intersection et la soustraction. Ces expressions retournent un résultat de type ensemble.

| N° | Désignation | Expression B | Contrainte OCL |
|---|---|---|---|
| 1 | Ensemble | $\{e_1,..., e_n\}$ | **Set**$\{e_1,.., e_n\}$ |
| 2 | Union | $ens \cup ens1$ | *ens*→**union(***ens1***)** |
| 3 | Intersection | $ens \cap ens1$ | *ens*→**intersection (***ens1***)** |
| 4 | Différence | $ens - ens1$ | *ens-ens1* |
| 5 | Union d'ensemble avec élément | $ens \cup \{elem\}$ | *ens*→**including (***elem***)** |
| 6 | Soustraction d'élément d'un ensemble | $ens - \{elem\}$ | *ens*→**excluding (***elem***)** |

TAB. 7.3 – Equivalence en OCL des expressions ensemblistes

Le tableau 7.4 énumère les règles de traduction des opérateurs de restriction sur les domaines et codomaines des relations sachant que dom et ran retournent respectivement le domaine (*e_R_D*) et le codomaine (*e_R_C*). Les autres opérateurs retournent des relations amoindries de leur restriction. La relation utilisée dans ce tableau est typée à l'aide d'un ensemble pour le domaine ($X$) et d'un autre pour le codomaine ($Y$). Ces ensembles sont extrait en classes : *e_X* et *e_Y*.

Le dernier tableau 7.5 traite les règles de traduction des expressions relatives aux suites telles que la définition, les insertions en tête et en queue, les sélections en tête et en queue, les suppressions multiples d'éléments en tête et en queue ainsi que les opérateurs de concaténation et d'inversement. Les variables OCL suivit de la chaîne de caractères *@pre* dénote la valeur des variables avant l'exécution de l'opération.

| N° | Désignation | Expression B | Contrainte OCL |
|---|---|---|---|
| 1 | Image | $R[X]$ | $R$→**select(**$X$→**includes(**$e\_X$**)** →**collect(**$e\_Y$**)** |
| 2 | Image (singleton) | $R[\{xx\}]$ | $R$→**select(**$xx$=$e\_X$**)**→**collect(**$e\_Y$**)** |
| 3 | Domaine | dom($R$) | $R$→**collect(**$e\_R\_D$**)** |
| 4 | Codomaine | ran($R$) | $R$→**collect (**$e\_R\_C$**)** |
| 5 | Restriction sur le domaine | $X \lhd R$ | $R$→**select(**$X$→**includes(**$e\_X$**))** |
| 6 | Soustraction sur le domaine | $X \lhd\!\!\!- R$ | $R$→**select(**$X$→**excludes(**$e\_X$**))** |
| 7 | Restriction sur le codomaine | $R \rhd Y$ | $R$→**select(Y→ includes(**$e\_Y$**)** |
| 8 | Soustraction sur le codomaine | $R \rhd\!\!\!- Y$ | $R$→**select(**$Y$→ **excludes(**$e\_Y$**)** |
| 9 | Surcharge | $R1 \lhd\!\!\!+ R2$ | $(R1-(R1$→**select(**$R2$→**includes(**$e\_X$**))))** →**union(**$R2$**)** |

TAB. 7.4 – Modélisation en OCL des expressions B portant sur les relations

| N° | Désignation | Expression B | Contrainte OCL |
|---|---|---|---|
| 1 | Suite | $[elem_1,..., elem_n]$ | **Sequence**$\{elem_1,..., elem_n\}$ |
| 2 | Taille | size(*suite*) | *suite*→**size()** |
| 3 | Insertion en queue | $suite \leftarrow elem$ | *suite*→**append(***elem***)** |
| 4 | Insertion en tête | $elem \rightarrow suite$ | *suite*→**prepend(***elem***)** |
| 5 | Premier élément | first (*suite)* | *suite*→**first()** |
| 6 | Dernier élément | last(*suite*) | *suite*→**last()** |
| 7 | Tête | front(*suite*) | *suite*→**subSequence(***1,* *suite*@**pre**→**size()**-*1***)** |
| 8 | Queue | tail(*suite*) | *suite*→**subSequence(***2,* *suite*@**pre**→**size())** |
| 9 | Restriction à la queue | $suite \downarrow n$ | *suite*→**subsequence(***n+1, suite*→**size())** |
| 10 | Restriction à la tête | $suite \uparrow n$ | *suite*→**subSequence(***1, n***)** |
| 11 | Concaténation | $suite1 \frown suite2$ | *suite1*→**union(***suite2***)** |
| 12 | Inverse | rev(*suite*) | **Sequence**$\{1..suite$→**size()**$\}$ →**iterate(***index ;s :T=***Sequence**$\{\}|$ *s*→**prepend(***suite*→**at(***index***)))** |

TAB. 7.5 – Equivalence en OCL des expressions sur les suites

### 7.2.3 L'exemple du Scheduler

L'exemple choisi est celui d'un gestionnaire de processus (la spécification B est présentée figure 7.1 et la modélisation UML figure 7.2). Celui-ci a pour but d'associer une ressource (un processeur par exemple) à un ensemble de processus. Un processus peut être dans les états

MACHINE
*Scheduler*
SETS
*PID*
CONCRETE_VARIABLES
*active, ready, waiting*
INVARIANT
$active \subseteq PID \wedge$
$ready \subseteq PID \wedge$
$waiting \subseteq PID \wedge$
$ready \cap waiting = \emptyset \wedge$
$(active = \emptyset \text{ or } (active \not\subset ready \cup waiting)) \wedge$
card $(active) \leq 1 \wedge$
$active = \emptyset \Rightarrow ready = \emptyset$
INITIALISATION
...
END

FIG. 7.1 – Extrait de la spécification B de l'exemple du *Scheduler*

suivants :
- waiting : le processus est présent mais non candidat à l'attribution de la ressource,
- ready : le processus est prêt à prendre la ressource,
- active : le processus est possesseur de la ressource.

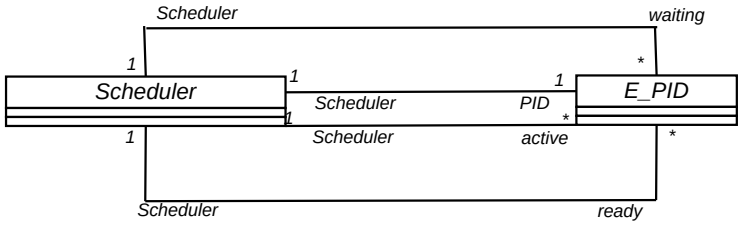


FIG. 7.2 – Modélisation en diagramme de classes de la machine *Scheduler*

L'invariant est composé de prédicats reliés par le connecteur $\wedge$. On applique donc la règle 1 du tableau 7.1 sur les conjonctions. Le prédicat active $\subseteq$ PID est transformé par la règle 12 du tableau 7.1 en *PID*→**includesAll(***active***)**. La même règle s'applique sur les deux prédicats suivants. L'expression d'intersection de deux ensembles égaux à l'ensemble vide en B : ready $\cap$ waiting $= \emptyset$, est représentée en OCL par la contrainte *ready*→**intersection(***waiting***)**→**isEmpty()** obtenue en appliquant la règle 3 du tableau 7.3 et la règle 20 du tableau 7.1. Les autres prédicats s'obtiennent de la même façon en recherchant les règles à appliquer. On obtient ainsi la contrainte finale de la figure 7.3.

**context** *Scheduler* **inv :**
*PID*→ **includesAll (***active***)**
**and** *PID*→**includesAll(***ready***)**
**and** *PID*→**includesAll(***waiting***)**
**and** *ready*→**intersection(***waiting***)**→**isEmpty()**
**and** *active*→**isEmpty() or not (***ready***)**→**union(***waiting***)**→**includesAll(***active***))**
**and** *active*→**size()**<= *1*
**and** *active*→**isEmpty() implies** *ready*→**isEmpty()**

FIG. 7.3 – Transformation d'une propriété invariante du *Scheduler*

### 7.2.4 Les problèmes rencontrés

#### 7.2.4.1 Le manque de concision d'OCL

Dans l'exemple du **scheduler**, il n'y a pas d'expressions et de prédicats très complexes mais si nous prenons un exemple comme le suivant ou ContenuEvacuation est une fonction partielle et AutorisationEvacuation une relation :

$\forall(xx, yy).((xx \mapsto yy) \in \text{ContenuEvacuation}$
$\Rightarrow yy \in \text{AutorisationEvacuation}[\{xx\}])$

La transformation en OCL (fig 7.4) est obtenue en utilisant les règles 6 et 10 du tableau 7.1 et la règle 2 du tableau 7.4.

Sur cet exemple on peut voir que si de nombreuses propriétés sont exprimables avec OCL, B reste beaucoup plus concis grâce à des opérateurs puissants. Outre cet aspect, la lisibilité, qui est certes un facteur beaucoup plus subjectif, nous paraît être à l'avantage de B par rapport à OCL.

*Evacuation*→**forAll(***ce : E_Evacuation* |
*AutorisationEvacuation*→**select(**
*ce.E_Evacuation_D = E_AutorisationEvacuation_D)*
→**collect(***E_AutorisationEvacuation_C***)**
→**includes(***ce.E_Evacuation_C***))**

FIG. 7.4 – Transformation en OCL d'une propriété invariante complexe

#### 7.2.4.2 Les expressions B non traduisibles en OCL

Premièrement nous avons omis de manière volontaire les expressions d'arbres et records qui sont peu utilisées en B et n'ont pas d'équivalent direct dans les structures OCL. Deuxièmement certaines expressions B sur les relations ne sont pas directement traduisibles en OCL avec le diagramme de classes par notre approche :

- Lorsqu'on utilise une relation ou une fonction sans l'avoir nommée dans l'invariant. Par exemple, dans le prédicat $x \in A \rightarrow (B \leftrightarrow C)$, la relation $B \leftrightarrow C$ pose un problème : il faudrait un nom pour au moins créer la classe représentant les couples.

– lorsqu'on utilise des opérateurs ou des fonctions B qui créent des relations qui n'existaient pas dans le diagramme de classes. On peut citer l'identité, l'inverse, les projections, le produit direct, le produit parallèle, la composition de relations.

La solution consiste à modifier le diagramme de classes en ajoutant les classes et les associations nécessaires et en choisissant des noms interactivement ou automatiquement pour ces nouveaux éléments. On pourrait limiter ce problème (mais pas l'éviter) en imposant des règles d'écriture sur les spécifications B.

## 7.3 Réécriture en OCL des opérations B

### 7.3.1 Présentation des opérations B

Nous nous intéressons ici à la traduction en OCL de la clause INITIALISATION et des opérations déclarées dans la clause OPERATIONS. Une opération est constituée d'une en-tête et d'un corps comme ci-dessous :

ListeDeRésultats $\leftarrow$ nomOpération (ListeDeParamètres) $\widehat{=}$
corps

Le corps des opérations et l'initialisation sont exprimés à l'aide de substitutions qui sont des notations mathématiques permettant de modéliser la transformation de prédicats. L'en-tête comprend une liste optionnelle de résultats typés implicitement dans le corps de l'opération et une liste optionnelle de paramètres typés explicitement s'ils existent grâce à la substitution pré-condition PRE *Prédicat* THEN *Substitutions* END. Celle-ci permet aussi de fixer les pré-conditions sous lesquelles une opération est appelée. Cette substitution constitue le corps de l'opération B et les autres substitutions sont imbriquées dans la partie *Substitutions*. Le typage des paramètres et les pré-conditions sont effectués dans la partie *Prédicat*.

L'exemple du **Scheduler** de la figure 7.5 permet de voir l'initialisation et deux opérations (*creer* et *activer*).

### 7.3.2 Transformation en OCL d'une opération B

#### 7.3.2.1 Principe général de transformation

A chaque opération d'une machine abstraite B est associée une opération de la classe représentant la machine. L'initialisation est représentée par une méthode spécifique appelée **INITIALIZATION**. A une opération d'une classe, OCL permet d'associer une pré-condition et une post-condition dont la syntaxe est la suivante :

**context** *nomClasse : :nomOpération(ListeDeparamètres : Types) : typeResultat*
**pre :** *pré-condition*
**post :** *post-condition*

La pré-condition OCL pour une opération est obtenue à partir du prédicat de la substitution pré-condition. La post-condition est obtenue en deux étapes :

1. On calcule à partir des substitutions B la post-condition B par le mécanisme de transformation en prédicat avant-après [3] paragraphe 6.3.3 et 7.1.1.
2. La post-condition B est réécrite en OCL à l'aide du mécanisme décrit en section 7.3.2.2.

```
MACHINE
    Scheduler
    ...
INITIALISATION
    active, ready, waiting := ∅, ∅, ∅
OPERATIONS
    creer(pp) ≙
        PRE
            pp ∈ PID ∧
            pp ∉ active ∧
            pp ∉ (ready ∪ waiting)
        THEN
            waiting := waiting ∪ {pp}
        END;
    activer(qq) ≙
        PRE
            qq ∈ waiting
        THEN
            waiting := waiting - {qq} ||
            IF (active = ∅) THEN
                active := {qq}
            ELSE
                ready := ready ∪ {qq}
            END
        END;
    ...
END
```

FIG. 7.5 – Autre extrait de la spécification du *Scheduler*

**context** *Scheduler : :INITIALIZATION()*
**post :***active*→**isEmpty() and** *ready*→**isEmpty() and** *waiting*→**isEmpty()**
**context** *Scheduler : :creer(pp : Elt_PID)*
**pre :** *active*→**excludes (***pp***)**
**and** *ready*→**union(***waiting***)**→**excludes(***pp***)**
**post :** *waiting* = *waiting*@**pre**→**including(***pp***)**
**context** *Scheduler : :activer(qq : Elt_PID)*
**pre :** *waiting*→**includes(***qq***)**
**post :** *waiting* = *waiting*@**pre**→**excluding(***qq***)**
**and if** *active*@**pre**→**isEmpty() then**
*active* = **Set**{*qq*}
**else** *ready* = *ready*@**pre**→**including(***qq***)**
**endif**

FIG. 7.6 – Contrainte OCL extraite de l'exemple du *Scheduler*

Les contraintes obtenues sur l'exemple du scheduler pour l'initialisation et les opérations *creer* et *activer* sont les suivantes :

En réalité, les post-conditions ont été obtenues de façon intuitive. Pour l'opération *activer*, une post-condition équivalente est calculée en 7.3.2.2.

Remarque : En B, une opération peut retourner plusieurs résultats. Dans la traduction UML-OCL, le premier résultat est considéré comme le résultat de l'opération et les autres sont passés en paramètres avec comme type de passage **out**.

#### 7.3.2.2 Exemple d'obtention d'une post-condition à partir des substitutions

On reprend ici le corps de l'opération *activer* et on applique les règles définies dans [3] pour trouver la post-condition B. Les variables primées des prédicats avant/après extrait des substitutions B dénotent la valeur de la variable après l'exécution des substitutions. En OCL, on désigne la valeur avant en suffixant la variable par **@pre**.

prd(*waiting* := *waiting* - {*qq*} ‖
IF *(active* = ∅*)* THEN *active* := {*qq*}
ELSE *ready* := *ready* ∪ {*qq*}
END)
⇔
prd(*waiting* := *waiting* - {*qq*}) ∧
prd(IF *(active* = ∅*)*
THEN *active* := {*qq*}
ELSE *ready* := *ready* ∪ {*qq*}
END)
⇔
waiting' = waiting - {qq}) ∧
prd((active = ∅ ⟹ active := {qq})
[] (¬ (active = ∅) ⟹ ready := ready ∪ {qq})

)

où $P \Longrightarrow S$ signifie $P$ *guards* $S$ tel que $P$ est un prédicat de précondition et $S$ une substitution.

$\Leftrightarrow$

waiting' = waiting - {qq}) $\wedge$
(prd(active = $\emptyset \Longrightarrow$ active := {qq})
$\vee$ prd($\neg$ (active = $\emptyset$) $\Longrightarrow$ ready := ready $\cup$ {qq})
)

$\Leftrightarrow$

waiting' = waiting - {qq}) $\wedge$
((active = $\emptyset \wedge$ prd(active := {qq}))
$\vee$ ($\neg$ (active = $\emptyset$) $\wedge$ prd(ready := ready $\cup$ {qq}))
)

$\Leftrightarrow$

waiting' = waiting - {qq}) $\wedge$
((active = $\emptyset \wedge$ prd(active := {qq}))
$\vee$ ($\neg$ (active = $\emptyset$) $\wedge$ prd(ready := ready $\cup$ {qq}))
)

$\Leftrightarrow$

waiting' = waiting - {qq}) $\wedge$
((active = $\emptyset \wedge$ active' = {qq}))
$\vee$ (active $\neq \emptyset \wedge$ ready' = ready $\cup$ {qq}))
)

La contrainte OCL se déduit de cette post-condition :

*waiting* = *waiting*@**pre**→**excluding(***qq***) and**
*((active*@**pre**→**isEmpty() and** *active* = **Set**{*qq*}*)*
**or** *(active*@**pre**→**notEmpty() and** *ready* = *ready*@**pre**→**including(***qq***))**
**)**

# Chapitre 8

# Extraction de diagrammes d'états-transitions

## 8.1 Introduction

Dans ce chapitre nous présentons les différents schémas de traduction de B vers les diagrammes d'états-transitions étendus à partir des publications [27, 49, 99]. Ces schémas concernent autant le B logiciel que le B système car la structure des *machines* est la même dans les deux formes de B. Nous définissons dans la section suivante les limites de cette traduction ainsi que la grammaire du noyau du langage B utilisé. Par la suite nous énonçons les règles de production du diagramme d'états-transitions. Enfin nous terminerons par une discussion à propos des choix de concaténation des différents diagrammes produits puis sur leur interprétation.

## 8.2 Limites de l'extraction

Nous ne traduisons que les parties significatives de la spécification B. Cette traduction ne s'applique qu'aux spécifications abstraites (machines abstraites et raffinements) possédant des variables scalaires typées à l'aide d'ensembles énumérés. Par conséquent les variables représentant des relations, fonctions, bijections, injections et surjections en sont exclues. Nous nous plaçons dans un cadre où la spécification B a un modèle à nombre d'états fini. En effet les diagrammes d'états-transitions de la notation UML ne permettent pas la représentation des systèmes infinis contrairement à certaines notations algébriques.

## 8.3 Grammaire du noyau du langage utilisé

Cette traduction s'applique aux machines abstraites et aux raffinements de la spécification B du système. La grammaire suivante détermine le noyau du langage B accepté pour notre traduction.

Spec : := Machine | Raffinement

Machine : := MACHINE Ident
Corps

Raffinement : := REFINMENT Ident REFINES Ident
Corps

Corps : := SETS Defs_Sets( ;Defs_Sets)*
VARIABLES Ident(,Ident)*
INVARIANT Ident ∈ Ident (∧ Ident ∈ Ident)*
INITIALISATION Affectations
OPERATIONS Operation ( ; Operation)*
END

Operation : := [Ident (,Ident)* ←] Ident [(Ident(,Ident)*)] ≙ Action

Defs_Sets : := Ident = { Ident(, Ident )+}

Affectation : := Ident := Expression |
Ident(, Ident )$^n$ := Expression (, Expression )$^n$ |
Ident :∈ Expression |
Ident :( Expression ) |
Skip

Action : := BEGIN Affectations END
| PRE Predicat ( ∧ Predicat )* THEN Affectations END
| SELECT Predicat ( ∧ Predicat )* THEN Affectations
( WHEN Predicat ( ∧ Predicat )* THEN Affectations )*
[ ELSE Affectations ] END
| ANY Ident (, Ident )* WHERE
Predicat ( ∧ Predicat )* THEN Affectations END
| CHOICE Affectations ( OR Affectations )* END
| IF Predicat ( ∧ Predicat )* THEN Affectations
( ELSIF Predicat ( ∧ Predicat )* THEN Affectations )*
[ ELSE Affectations ] END
| CASE Ident OF
EITHER Expression (,Expression )* THEN Affectations
( OR Expression (,Expression )* THEN Affectations )*
[ ELSE Affectations ] END
END

Affectations : := Affectation ( Separateur Affectation )*

*Expression* et *Prédicat* ne sont pas définis dans cette grammaire mais le sont dans les annexes du manuel de référence du langage B [93] section B.1.6 et B.1.7. *Separateur* est l'opérateur '||' dans les machines et un des opérateurs '||' ou ';' dans les raffinements.

## 8.4 Principes généraux de traduction

Dans cette section, nous explicitons les principes généraux utilisés pour l'extraction de spécifications B en diagrammes d'états-transitions notamment leur structure. Ensuite nous définissons les principes d'extraction des substitutions. Les substitutions du langage B (paragraphe 2.3.5) sont partagées en deux groupes distincts :

- les substitutions simples,
- les substitutions de composition.

En règle générale, les substitutions simples définissent l'état cible de la ou des transition(s) tandis que les substitutions de composition fixent l'état source de la ou des transition(s) par les gardes ou conditions selon le type de transition. Dans ce chapitre on utilisera les variables $x$ et $y$ et les valeurs qu'elles pourront prendre respectivement $vx_1$, ..., $vx_n$ et $vy_1$, ..., $vy_n$.

## 8.5 La structure

MACHINE *M1*
VARIABLES *x,y*
...
END

FIG. 8.1 – Des variables

Comme il est écrit précédemment le principal élément de la méthode B est la machine abstraite. Celle-ci est traduite par un super-état qui prend le nom de cette machine. Dans l'exemple suivant, la machine (fig 8.1) se nomme *M1* et le super-état résultant de son extraction (fig 8.2) prend le même nom. Ensuite on ajoute des variables à la machine (fig 8.1). Chaque variable correspond elle aussi à un super-état. Comme pour les machines ce super-état prend le nom de la variable. Il est inséré dans le super-état de la machine où la variable est définie (fig 8.2).

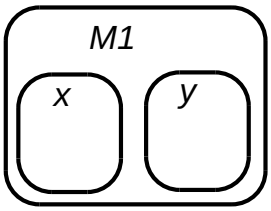


FIG. 8.2 – Extraction des variables

Enfin il faut typer les variables de la spécification. Comme nous l'avons défini dans la section 8.3, on ne considère que les variables typées à l'aide d'ensembles énumérés. Ils sont définis par la clause SETS (fig 8.3). On type ensuite les variables par la clause INVARIANT. La variable $x$ est typée à partir de l'ensemble énuméré $X$.

Au niveau du diagramme d'états-transitions, on traduit le typage de la variable $x$ par le fait que $x$ peut prendre toutes les valeurs comprises dans l'ensemble $X$. Sachant qu'une valeur

MACHINE *M1*
SETS $X=\{vx_1,...,vx_n\}$
VARIABLES $x$
INVARIANT $x \in X$
...
END

FIG. 8.3 – Une variable typée à l'aide d'un ensemble énuméré

de variable correspond à un état du diagramme d'états-transitions on crée un état par valeur $vx_1$, ..., $vx_n$ (fig 8.4).

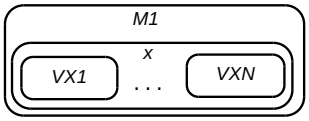


FIG. 8.4 – Extraction d'une variable typée à l'aide d'un ensemble énuméré

## 8.6 Les substitutions simples

Dans cette section on considérera une seule variable. Les substitutions simples ne déterminent que l'état cible. Les substitutions simples sont :

- Identité *skip*
- Devient tel que :()
- Devient élément de :∈
- Devient égal simple et multiple :=

Les substitutions *Devient tel que, Devient élément de, Devient égal simple* et *Devient égal multiple* s'expriment suivant la grammaire définie précédemment. Dans les règles définissant ces substitutions apparaît une expression qu'il faut interpréter pour connaître le ou les état(s) cible(s) correspondant(s) à l'extraction de la substitution. L'interprétation des expressions utilisées par *Devient élément de* et *Devient tel que* peut faire apparaître plusieurs valeurs tandis que celle des expressions utilisées par *Devient égal simple* et *multiple* fait apparaître une seule valeur. Selon leur sémantique (tableau 2.2) il existe trois catégories par rapport à leurs effets sur l'état du système.

1. Une première catégorie composée de la substitution *identité* n'a aucun effet sur l'état du système. Néanmoins elle correspond à une transition. Elle sera donc traduite par une transition réflexive sur l'état courant du système (fig 8.5).
2. Une deuxième catégorie est composée des substitutions non déterministes telles que *devient tel que* et *devient élément de*. Le nombre d'états cibles est supérieur à un car elles sont non déterministes. Ces substitutions utilisent des expressions :
   - *x :(Expression)*
   - *x :∈ Expression*

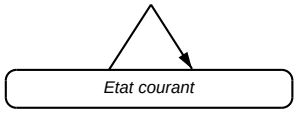


FIG. 8.5 – Extraction de la substitution *Identité*

L'interprétation de ces expressions peut donner plusieurs valeurs. On considérera l'ensemble des états $E_{cible}$ = {*VX1, ..., VXN*} traduits des valeurs résultantes de l'interprétation. Ces substitutions sont traduites (fig 8.6) par autant de transitions que d'états de $E_{cible}$ prenant pour origine l'état courant.

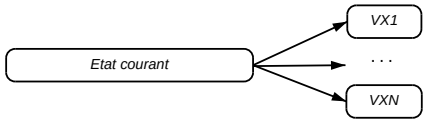


FIG. 8.6 – Extraction des substitutions simples non déterministes

3. Enfin la dernière catégorie qui correspond aux substitutions simples déterministes est composée des substitutions *Devient égal simple* et *Devient égal multiple*. La différence entre ces deux substitutions est le nombre de variables modifiées. En effet la substitution *Devient égal multiple* effectue plusieurs substitutions *Devient égal simple* simultanément. Dans cette catégorie l'état cible de la transition est clairement défini car l'interprétation des expressions donne un résultat ne comprenant qu'une seule valeur :
   – *x := Expression*
   – *x, y := Expression, Expression*

   L'extraction de ces substitutions est réalisée de la manière suivante. La substitution *Devient égal simple* (fig 8.7) est traduite par une transition partant de l'état courant, et qui se dirige vers un et un seul état cible, ici *VX1*, car elle est déterministe. La substitution *devient égal multiple* effectue plusieurs substitutions *Devient égal simple* simultanées donc son extraction fera apparaître autant de fois que nécessaire le schéma de l'extraction de la substitution *Devient égal simple* mais pour différentes variables car elle n'autorise pas l'affectation de la même variable plusieurs fois.

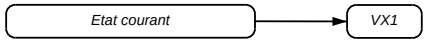


FIG. 8.7 – Extraction de la substitution *Devient égal simple*

## 8.7 Initialisation, opérations et événements

L'initialisation est une opération particulière qui est la première exécutée. En B, l'initialisation et les opérations sont modélisées en utilisant les substitutions généralisées. Toutefois il existe une différence en ce qui concerne les substitutions conditionnelles concernant une valeur de variable. En effet une condition se détermine par rapport à la valeur d'une variable or avant l'initialisation la valeur des variables n'est pas définie donc on ne peut pas avoir ce

genre de substitution à l'initialisation en utilisant l'opérateur '||'. Cependant leur utilisation est possible en utilisant le *Séquencement* exprimé à l'aide de l'opérateur ';', mais les valeurs des variables à la fin de l'initialisation seront toujours les mêmes car l'affectation sera réalisée en fonction de la valeur d'une variable qui sera toujours la même. On peut mettre tout type de substitution dans les opérations si l'utilisation est autorisée par rapport aux restrictions dues au type de composant B (MACHINE ou REFINEMENT).

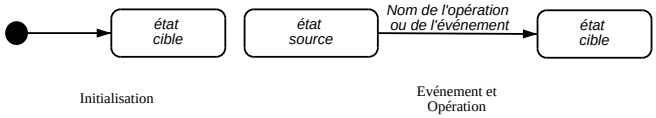


FIG. 8.8 – Principe général de l'extraction

Une transition part d'un état source vers un état cible. L'état source est défini par la valeur de la variable avant l'exécution de l'opération ou de l'événement. Avant l'initialisation aucune valeur de variable n'est définie. L'initialisation est représentée par un disque noir plein, qui représente l'état courant, suivi d'une flèche (fig 8.8) qui se dirige vers un état cible. Le ou les états qui sont concernés par l'issue de l'initialisation sont définis par la ou les valeurs que prend la variable du diagramme d'états-transitions selon que la substitution est déterministe ou non.

Les événements et les opérations sont représentés par une flèche (transition) (fig 8.8) qui part de l'état source (valeur de la variable avant son exécution) et qui se dirige vers l'état cible (valeur de la variable après son exécution). Cette flèche prend le nom de l'événement ou de l'opération auquel elle se rapporte.

## 8.8 Les substitutions de composition

Dans cette section, on utilisera la substitution simple *Devient égal simple* pour simplifier les règles de production de la traduction. Il est néanmoins possible d'utiliser n'importe quelle substitution simple. Les règles de production seront modifiées de manière à remplacer la seule transition de la substitution *Devient égal simple* par la ou les transition(s) correspondante(s) à la traduction de la substitution utilisée.

### 8.8.1 Substitution *Bloc*

La substitution *Bloc* s'exécute sans aucune condition (fig 8.9).

$OP \mathrel{\widehat{=}}$ BEGIN $x := vx_i$ END

FIG. 8.9 – La substitution *Bloc*

L'extraction fait apparaître une transition depuis chaque état de la variable, car cette substitution ne possède aucune restriction, et dirigée vers l'état qui correspond à la valeur $vx_i$ (fig 8.10).

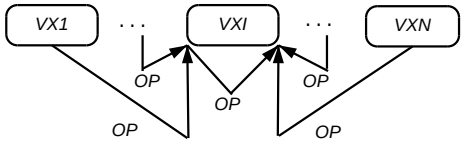


FIG. 8.10 – Extraction de la substitution *Bloc*

### 8.8.2 Substitution *Sélection*

La substitution *Sélection* utilise un prédicat $P$ comme condition d'exécution (fig 8.11). Si ce prédicat est faux, aucune action n'est exécutée.

$OP \mathrel{\widehat{=}}$ SELECT $P$ THEN $x := vx_i$ *END*

FIG. 8.11 – La substitution *Sélection*

Etant donné que l'on ne connaît pas la nature du prédicat $P$ on le fait apparaître dans les étiquettes de transition de l'extraction par *OP[P]* (fig 8.12).

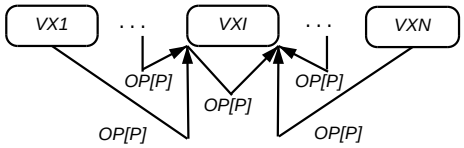


FIG. 8.12 – Extraction de la substitution *Sélection*

Les branches WHEN ... THEN sont traduites par une transition supplémentaire.

### 8.8.3 Substitution *Précondition*

La substitution *Précondition* utilise un prédicat $P$ appelé pré-condition. Dans le cas ou cette pré-condition est vraie la substitution est exécutée ; dans le cas contraire (la pré-condition notée $P$ n'est pas vraie et devient $\neg P$) on ne connaît pas le résultat de l'opération (fig 8.13). Elle est indéfinie mais elle est exécutée.

$OP \mathrel{\widehat{=}}$ *PRE* $P$ THEN $x := vx_i$ END

FIG. 8.13 – La substitution *Précondition*

L'extraction fait apparaître le principe cité en début de paragraphe. Dans le cas ou $P$ est vraie les transitions sont étiquetées par *OP[P]* et à l'inverse par *OP[¬P]* (fig 8.14). Pour augmenter la lisibilité de la figure 8.14, les seules transitions représentées sont celles entre les états visibles qui sont *VX1, VXI* et *VXN*. Les états représentés par ... ont une transition étiquetée *OP[P]* dirigée vers *VXI* et des transitions étiquetées *OP[¬P]* vers tous les états du diagramme.

### 8.8.4 Substitution *Choix borné*

La substitution *Choix borné* est non déterministe (fig 8.15).

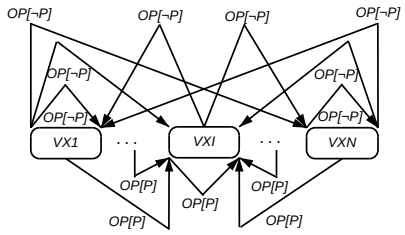


FIG. 8.14 – Extraction de la substitution *Précondition*

$OP \widehat{=}$ CHOICE $x := vx_i$ ... OR $x := vx_j$ END

FIG. 8.15 – La substitution *Choix borné*

L'extraction fait apparaître deux groupes distincts de transitions, celles issues de chaque état et qui convergent vers l'état *VXI* et celles vers *VXJ* (fig 8.16). Il existe des transitions issues des états représentés par ... mais elles ne sont pas représentées pour rendre la figure 8.16 plus lisible.

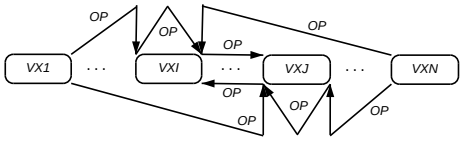


FIG. 8.16 – Extraction de la substitution *Choix borné*

### 8.8.5 Substitution *Condition*

A la différence de la substitution *Sélection* la substitution *Condition* provoque une exécution dans tous les cas. Lorsque le prédicat $P$ est faux alors la branche *ELSE* est exécutée. Si elle n'est pas définie (fig 8.17) une branche *ELSE* implicite est exécutée comme dans le tableau 8.1 selon [48].

| Substitution *Condition* | Substitution *Sélection* |
|---|---|
| IF $P$ THEN $S$ END | SELECT $P$ THEN $S$<br>WHEN $\neg P$ THEN *Skip* END |

TAB. 8.1 – Extraction de la substitution *Condition* à l'aide de *Sélection*

$OP \widehat{=}$ IF $P$ THEN $x := vx_i$ END

FIG. 8.17 – La substitution *Condition*

L'extraction (fig 8.18) met en évidence deux groupes de transitions, les transitions convergeant sur l'état *VXI* étiquetées par *OP[P]* et le groupe des transitions réflexives étiquetées par *OP[¬P]* traduites de la branche *ELSE* implicite.

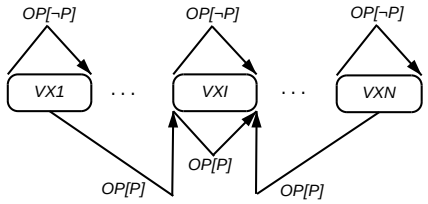


FIG. 8.18 – Extraction de la substitution *Condition*

L'extraction de la branche *ELSIF ... THEN* fait apparaître une transition de la même forme que celle traduite depuis la branche *IF ... THEN*.

### 8.8.6 Substitution *Condition par cas*

La substitution *Condition par cas* possède les mêmes caractéristiques que la substitution *Condition*, elle provoque une exécution dans tous les cas. Elle peut être traduite elle aussi à l'aide de la substitution *Sélection* comme dans le tableau 8.2.

| Substitution *Condition par cas* | Substitution *Sélection* |
|---|---|
| CASE *E* OF | SELECT $E \in LC1$ THEN *S* |
| EITHER *LC1* THEN *S* | WHEN $E \in LC2$ THEN *T* |
| OR *LC2* THEN *T* END END | ELSE *Skip* END |

TAB. 8.2 – Extraction de la substitution *Condition par cas* à l'aide de *Sélection*

*OP* $\widehat{=}$ CASE *E* OF
EITHER *LC1* THEN $x := vx_i$
OR *LC2* THEN $x := vx_j$ END END

FIG. 8.19 – La substitution *Condition par cas*

L'extraction fait apparaître trois groupes de transitions. Un premier groupe est étiqueté par *OP[P1]* où $P1 = E \in LC1$ et converge vers l'état *VXI* et le deuxième vers *VXJ* et est étiqueté par *OP[P2]* où $P2 = E \in LC2$. Le troisième qui résulte de l'extraction de la branche implicite *ELSE* englobe les transitions étiquetées par *OP[¬P]* où $P = P1 \vee P2$ réflexives sur chaque état du diagramme (voir figure 8.19).

### 8.8.7 Substitution *Choix non borné*

La construction syntaxique et sémantique de la substitution *Choix non borné* est particulière (fig 8.21). En effet l'affectation n'est pas réalisée à l'aide d'une expression ni d'une valeur mais par un identifiant *Ident* dont la valeur $vx_{Ident_P}$ est définie en fonction du prédicat *P*. $vx_{Ident_P}$ est un choix non déterministe entre plusieurs valeurs possibles selon le prédicat *P*.

Dans l'extraction (fig 8.22), tant que l'on ne connaît pas la nature du prédicat *P* les transitions sont issues de tous les états et divergent vers tous les états. Néanmoins le prédicat

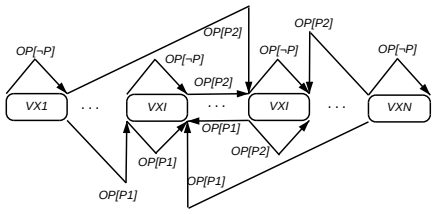


FIG. 8.20 – Extraction de la substitution *Condition par cas*

*OP* $\widehat{=}$ ANY *Ident*
WHERE *P* THEN *x* := *Ident* END

FIG. 8.21 – La substitution *Choix non borné*

*P* est une condition pour que la transition soit possible d'un état à un autre. Par conséquent les transitions sont étiquetées par ***OP[P]***.

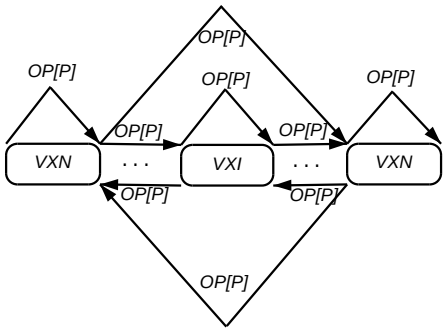


FIG. 8.22 – Extraction de la substitution *Choix non borné*

### 8.8.8 Simplifications par évaluation des expressions

Les diagrammes que l'on peut générer à partir de l'extraction précédente sont complexes. Il est possible de les simplifier pour obtenir une meilleure lisibilité de ceux-ci. Les prédicats maintenus tels quels dans les étiquettes de transitions peuvent être interprétés dans certains cas lorsqu'ils sont exprimés en fonction de la valeur initiale de la variable (avant transition) relative au diagramme d'états-transitions considéré pour les substitutions suivantes :

- *Sélection*
- *Précondition*
- *Condition*
- *Condition par cas*
- *Choix non borné*

Nous cherchons à interpréter *P* relativement à l'état source de la transition. Les expressions interprétables de *P* font partie de la liste non exhaustive suivante :

- $x = vx_i$
- $x \neq vx_i$

- $x \in \{vx_i, ..., vx_j\}$
- $x \notin \{vx_i, ..., vx_j\}$

$P$ peut être vrai dans un état *VXI*, les transitions étiquetées par *OP[¬P]* issues de l'état *VXI* disparaissent et inversement lorsqu'il est faux les transitions étiquetées *OP[P]* (issues de *VXI*) disparaissent. Cependant le prédicat $P$ n'est pas obligatoirement interprétable dans sa totalité. Il existe quatre cas de figure où une partie de $P$ est interprétable sachant que généralement une transition dont la condition est $P \mathrel{\widehat{=}} (P1 \vee P2)$ donne deux transitions distinctes l'une avec la condition $P1$ et l'autre avec $P2$ :

- $P \mathrel{\widehat{=}} (P1 \wedge P2)$ où $P2 \mathrel{\widehat{=}} vrai \Rightarrow P \mathrel{\widehat{=}} (P1 \wedge vrai) \Rightarrow P \mathrel{\widehat{=}} P1$
- $P \mathrel{\widehat{=}} (P1 \vee P2)$ où $P2 \mathrel{\widehat{=}} vrai \Rightarrow P \mathrel{\widehat{=}} (P1 \vee vrai) \Rightarrow P \mathrel{\widehat{=}} vrai$
- $P \mathrel{\widehat{=}} (P1 \wedge P2)$ où $P2 \mathrel{\widehat{=}} faux \Rightarrow P \mathrel{\widehat{=}} (P1 \wedge faux) \Rightarrow P \mathrel{\widehat{=}} faux$
- $P \mathrel{\widehat{=}} (P1 \vee P2)$ où $P2 \mathrel{\widehat{=}} faux \Rightarrow P \mathrel{\widehat{=}} (P1 \vee faux) \Rightarrow P \mathrel{\widehat{=}} P1$

L'évaluation pourrait être réalisée à l'aide d'un outil comme l'animateur faisant partie de l'outil BZ Testing Tools[1] [65] développé au sein du LIFC. Cet animateur [25, 39] permet d'évaluer d'une part les gardes et conditions des substitutions et d'autre part les expressions utilisées dans les substitutions simples d'affectation déterministes ( :=) et non déterministes ( :∈ et :()).

Une autre simplification est la suppression des transitions réflexives autour des états car elles ne modifient pas l'état du système. Elle permet une meilleure lisibilité du diagramme. Néanmoins il ne faut en aucun cas supprimer les transitions réflexives extraites depuis les branches ELSE implicites des substitutions *Condition* et *Condition par cas*.

Nous allons appliquer un exemple d'interprétation avec la substitution *Sélection*. On considérera le prédicat $P \mathrel{\widehat{=}} x = vx_1$. Le diagramme de la figure 8.12 devient alors celui de la figure 8.23.

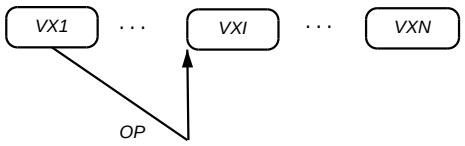


FIG. 8.23 – Extraction de la substitution *Sélection* après interprétation de $P$

### 8.8.9 Variables concurrentes

Les variables sont dites concurrentes quand elles appartiennent à la même machine et que l'une fait partie de la condition de transition de l'autre. Dans la spécification suivante (fig 8.24) la valeur de la variable $x$ est modifiée selon une substitution conditionnelle relative à $y$.

Dans le diagramme d'états-transitions de la variable $x$ (fig 8.25) correspondant à la spécification (fig 8.24), $y = Y1$ est une condition elle est donc entre crochets. Le fait que la condition de transition est en fonction de $y$ est symbolisée par IN entre crochet.

### 8.8.10 Paramètres

En B les paramètres d'une opération sont définis entre parenthèses à la suite du nom de l'opération (fig 8.26). Les paramètres sont concrets car ils sont conservés lors des raffinements et de l'écriture de l'implémentation qui utilise uniquement des données concrètes.

[1] http ://lifc.univ-fcomte.fr/∼bztt/

MACHINE *M1*
SETS $X=\{VX1, ..., VXN\}$ ;$Y=\{VY1, ..., VYN\}$
VARIABLES *x,y*
INVARIANT $x \in X \wedge y \in Y$
INITIALISATION $x := VX1 \parallel y := VY1$
OPERATIONS
$OP \widehat{=}$ IF $x = VX1 \wedge y = VY1$ THEN $x := VX2 \parallel y := VY2$ END
END

FIG. 8.24 – Deux variables concurrentes

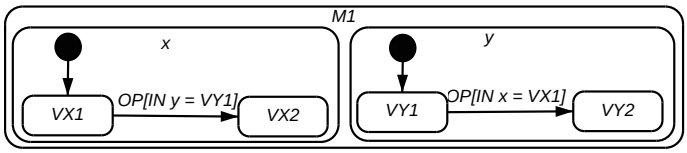


FIG. 8.25 – Traduction des variables concurrentes

...
*OP(Param)* $\widehat{=}$ ...

FIG. 8.26 – Une opération avec paramètre

On retrouve ce paramètre défini de la même manière entre parenthèses à la suite du nom de la transition sur le diagramme d'états-transitions (fig 8.27).

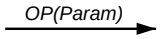


FIG. 8.27 – Traduction des paramètres

## 8.9 Calcul de diagrammes extraits de substitutions imbriquées

Nous considérons les branches conditionnelles $CondP_x$ d'une substitution principale SUB_PRINCIPALE_$B_x$ à l'intérieur desquelles sont insérées des substitutions de composition secondaires SUB_SECONDAIRE_$B_x$ avec conditions $CondS_x$ (fig 8.28) où $x$ représente le numéro de branche.

Ce sont les substitutions simples $SUB\_SIMPLE_x$ qui déterminent l'état cible de la transition. Or chaque état cible déterminé par une substitution simple correspond à un groupe de transitions dont l'état cible est défini par les substitutions de composition. Par conséquent il existe un groupe de transitions par substitution simple contenu dans les substitutions de composition imbriquées. Les conditions des transitions sont calculées par concaténation des conditions des substitutions principales et secondaires où se situe la substitution simple. Le groupe de transitions correspondant à la substitution simple $SUB\_SIMPLE_1$ a donc pour

*OP*$\widehat{=}$ SUB_PRINCIPALE_$B_1$ *CondP*$_1$ THEN
SUB_SECONDAIRE_$B_1$ *CondS*$_1$ THEN *SUB_SIMPLE*$_1$
SUB_SECONDAIRE_$B_2$ *CondS*$_2$ THEN *SUB_SIMPLE*$_2$
END
END

FIG. 8.28 – Imbrication de substitutions de composition

condition *CondP*$_1 \wedge$ *CondS*$_1$. Le calcul des transitions du diagramme est réalisé de la manière décrite dans les paragraphes précédents selon la substitution principale en reproduisant le groupe obtenu autant de fois que de branches de la substitution secondaire sans oublier de calculer la condition de transition du groupe.

## 8.10 Calcul du diagramme significatif par sélection de variables

Chaque opération est traduite par un morceau de diagramme d'états-transitions relatif à une ou plusieurs variables. Pour obtenir le diagramme d'états-transitions complet d'une variable il suffit de concaténer tous les morceaux relatifs à la variable. Dans chaque sous-diagramme les transitions sont issues d'un état et se dirigent vers un autre. Il suffit de reproduire les transitions de chaque sous-diagramme sur le diagramme complet, par rapport aux états sources et cibles des transitions, pour concaténer les sous-diagrammes.
Lorsque plusieurs variables sont mises en relation dans l'invariant, il est intéressant à partir des diagrammes de ces variables d'établir un seul et unique diagramme. Ceci aura pour but de valider le cahier des charges en vérifiant la propriété considérée sur le diagramme résultat. On obtient le diagramme résultat en effectuant un produit synchronisé entre les diagrammes d'états-transitions des variables concernées.

On considère les automates suivants, $A_i$=<$S_i$, $s_{0_i}$, $E_i$, $R_i$, $L_i$> où :
- $S_i$ est l'ensemble des états
- $s_{0_i}$ est l'ensemble des états initiaux $s_{0_i} \in S_i$
- $E_i$ est l'ensemble des étiquettes de transition
- $R_i$ est la relation de transition définie par $R_i \subseteq S_i \times E_i \times C_i \times S_i$ où $C_i$ : := $ap_i$ | $C_i \vee C_i$ | $\neg$ $C_i$ avec $ap_i$ une proposition atomique relative à une valeur de variable de l'automate $A_i$.
- $L_i$ est le décor des états défini par $L_i : S_i \rightarrow 2^{AP_i}$ avec $AP_i$ un ensemble de propositions atomiques.

Définition de la relation de satisfaction d'une proposition d'état $p$ par un état $s$ est :
- $s \models p$ si et seulement si $p \in L(s)$
- $s \models p_1 \vee p_2$ si et seulement si $L(s) \models p_1$ ou $s \models p_2$
- $s \models \neg p$ si et seulement si $L(s) \not\models p$

On utilise produit synchronisé pour obtenir l'automate résultat noté : $A_1 \parallel A_2 \setminus$ Sync

Soit *Sync* un ensemble de synchronisation qui est un sous-ensemble de $E_1 \cup E_2 \cup E_1 \times E_2$. Le produit synchronisé des automates $A_1$ et $A_2$, noté $A_1 \parallel A_2 \setminus$ Sync est un automate <$S_1 \times S_2$, ($s_{0_1}$, $s_{0_2}$), Sync, R, L> où R est la relation de transition inclue dans ($S_1 \times S_2$) $\times$

Sync $\times$ ($S_1 \times S_2$) définie aussi :

- $(s_1, s_2) \xrightarrow{e_1} (s_{1'}, s_2) \in R$ si $s_1 \xrightarrow{e_1, INc_2} s_{1'} \in R_1$, $s_2 \vDash c_2$ et $e_1 \in$ Sync,
- $(s_1, s_2) \xrightarrow{e_2} (s_1, s_{2'}) \in R$ si $s_2 \xrightarrow{e_2, INc_1} s_{2'} \in R_2$, $s_1 \vDash c_1$ et $e_2 \in$ Sync,
- $(s_1, s_2) \xrightarrow{(e_1, e_2)} (s_{1'}, s_{2'}) \in R$ si $s_1 \xrightarrow{e_1, INc_2} s_{1'} \in R_1$, $s_2 \xrightarrow{e_2, INc_1} s_{2'} \in R_2$, $s_1 \vDash c_1$, $s_2 \vDash c_2$ et $(e_1, e_2) \in$ Sync.

Dans notre contexte *Sync* est défini implicitement par la synchronisation des actions de même nom de $A_1$ et $A_2$ :
Sync = $\{e_1 \mid e_1 \in E_1 \text{ et } e_1 \notin E_2\} \cup \{e_2 \mid e_2 \notin E_1 \text{ et } e_2 \in E_2\} \cup \{e \mid e \in E_1 \text{ et } e \in E_2\}$
Les actions qui n'appartiennent qu'à $E_1$ ou $E_2$ restent libres. Dans les figures pour simplifier, nous notons le couple *(e,e)* par *e*.

MACHINE *M1*
SETS *ETAT*={*ouvert,ferme*}
VARIABLES *v1,v2*
INVARIANT
*v1* $\in$ *ETAT* $\wedge$
*v2* $\in$ *ETAT* $\wedge$
*v1* = *ouvert* $\Rightarrow$ *v2* = *ferme* $\wedge$
*v2* = *ouvert* $\Rightarrow$ *v1* = *ferme*
INITIALISATION
*v1* := *ferme* || *v2* := *ferme*
OPERATIONS
*ouvrir_v1* $\widehat{=}$
SELECT *v2* = *ferme* THEN *v1* := *ouvert* END;
*ouvrir_v2* $\widehat{=}$
SELECT *v1* = *ferme* THEN *v2* := *ouvert* END;
*fermer* $\widehat{=}$
BEGIN *v1* := *ferme* || *v2* := *ferme* END
END

FIG. 8.29 – Exemple de machine

Soit la machine *M1* (fig 8.29) qui permet de gérer deux vannes d'alimentation *v1* et *v2*. Une règle de gestion (R) du cahier des charges stipule que *v1* et *v2* ne sont jamais ouvertes en même temps ce qui se traduit dans l'invariant par *v1* = *ouvert* $\Rightarrow$ *v2* = *ferme* $\wedge$ *v2* = *ouvert* $\Rightarrow$ *v1* = *ferme*. Les diagrammes des variables *v1* et *v2* respectivement $D_{v1}$ et $D_{v2}$ sont les suivants figure 8.30.

On calcule le diagramme de *M1*, $D_{M1}$ (fig 8.31) par le produit synchronisé des diagrammes notés || défini dans [83]. La synchronisation est réalisée à l'aide des couples d'événements de transitions qui apparaissent dans les deux diagrammes (*(fermer, fermer)*) qui sont mis en jeu par le produit synchronisé ainsi que les événements qui n'apparaissent que dans l'un des deux diagrammes (*ouvrir_v1, ouvrir_v2*) : $D_{M1} = D_{v1} \parallel D_{v2}/\{$ouvrir_v1, ouvrir_v2,

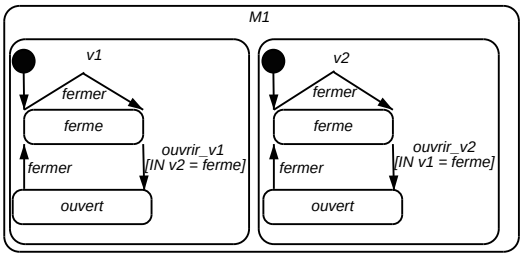


FIG. 8.30 – Automate des variables *v1* et *v2*

(fermer,fermer)}.

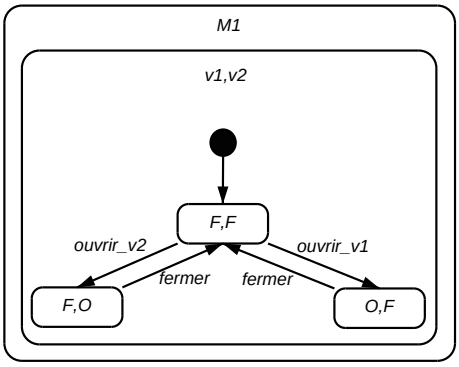


FIG. 8.31 – Automate final de la machine *M1*

On peut alors vérifier le respect de la règle de gestion (R) par le système. Il n'existe pas d'états où les deux vannes sont ouvertes en même temps.

## 8.11 L'exemple du Robot

L'exemple suivant est un cas d'étude [2] réalisé à partir d'une spécification écrite selon une approche B événementielle.

### 8.11.1 Présentation

Le système physique à piloter est schématisé dans la Fig 8.32. Il est composé de trois dispositifs :

- un dispositif d'arrivée des pièces appelé *Da*,
- un dispositif d'évacuation des pièces appelé *De*,
- un dispositif de transport des pièces, appelé *Dt*, qui est un bras muni d'une pince ; le bras dispose de trois mouvements (vertical, horizontal et en profondeur) ; la pince a également un mouvement d'ouverture et de fermeture.

Aucun des trois dispositifs ne peut contenir plus d'une pièce à la fois.

### 8.11.2 Principes de fonctionnement

Les pièces arrivent sur le dispositif *Da* et *Dt* effectue le chargement de ces pièces à partir de *Da* et les décharge sur *De*. Les pièces sont ensuite évacuées automatiquement. *Dt* ne peut

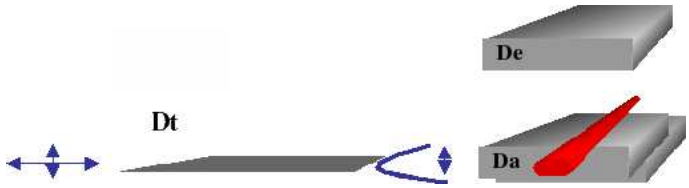


FIG. 8.32 – Le robot

charger une pièce que si *Dt* est libre (*vid*) et que *Da* est occupé (*occ*) (une pièce est arrivée). Il ne peut décharger une pièce sur le dispositif d'évacuation *De* que si celui-ci est libre (*vid*). Une pièce ne peut arriver sur un dispositif d'arrivée que si celui ci est libre (*vid*). Les mouvements du dispositif de transport sont :

- *Dt* décharge en position haute,
- *Dt* charge en position basse,
- *Dt* doit tenir une pièce pour monter,
- *Dt* doit être vide pour descendre.

Dans un souci de simplification, nous ne présentons pas la modélisation des mouvements de la pince ni ceux du dispositif de transport en profondeur.

### 8.11.3 Diagramme d'états transitions des machines abstraites

#### 8.11.3.1 Machine abstraite

La machine abstraite (fig 8.33) décrit le dispositif *Dt*. Dans la clause OPERATIONS nous trouvons les événements *Chgt* et *Dchgt* qui correspondent respectivement au chargement et déchargement de pièces par le dispositif *Dt*. Nous remarquons que l'événement *Chgt* fait passer *Dt* de l'état vide (*vid*) à l'état occupé (*occ*), par contre l'événement *Dchgt* réalise l'inverse.

MACHINE *Robot0*
SETS *ETAT*={*vid*,*occ*}
VARIABLES *dt0*
INVARIANT
    *dt0* ∈ {*vid*, *occ*}
INITIALISATION
    *dt0* := *vid*
OPERATIONS
    *Chgt* ≙ SELECT *dt0* := *vid* THEN *dt0* :=*occ* END
    *Dchgt* ≙ SELECT *dt0* := *occ* THEN *dt0* :=*vid* END
    *arr_p* ≙ *skip*
    *evac* ≙ *skip*
END

FIG. 8.33 – Machine abstraite

Le diagramme de la variable *dt0* (fig 8.38), est extrait à partir de trois éléments de la spécification B du robot (fig 8.33) qui sont :

– l'initialisation
– l'opération *Chgt*
– l'opération *Dchgt*

Le diagramme extrait de l'initialisation est celui de la figure 8.34.

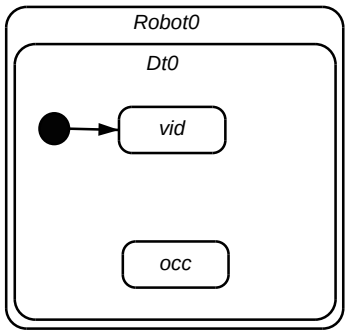


FIG. 8.34 – Diagramme d'états transitions du *Robot0* extrait de l'initialisation

Le diagramme extrait de l'opération *Chgt* sans simplification est celui de la figure 8.35.

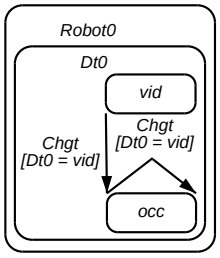


FIG. 8.35 – Diagramme d'états transitions du *Robot0* extrait de l'opération *Chgt*

On obtient le diagramme 8.36 après la simplification par suppression de la transition réflexive sachant que $P \mathrel{\widehat{=}} dt0 = vid$ est vrai dans l'état *vid*, la condition *[Dt0 = vid]* disparaît des transitions.

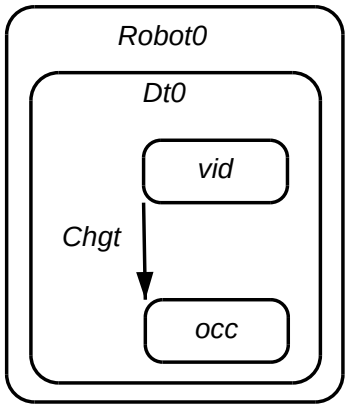


FIG. 8.36 – Diagramme extrait de l'opération *Chgt* après simplification

De la même manière on obtient le diagramme correspondant à l'opération *Dchgt* (fig 8.37).

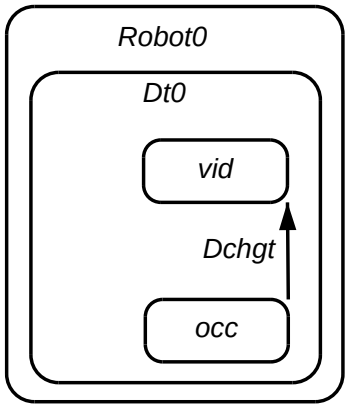


FIG. 8.37 – Diagramme extrait de l'opération *Dchgt* après simplification

Il reste à concaténer les trois diagrammes des figures 8.34, 8.36 et 8.37 pour obtenir le diagramme complet 8.38 de la spécification du robot 8.33.

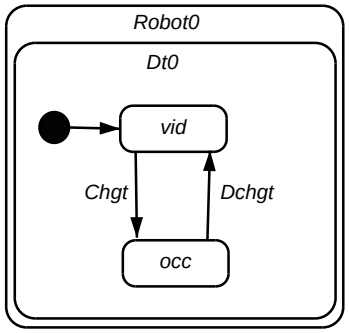


FIG. 8.38 – Diagramme d'états-transitions du *Robot0*

#### 8.11.3.2 Premier raffinement

Dans le premier raffinement (Fig 8.39), nous introduisons le dispositif d'arrivée représenté par la variable *Da1*. Cette variable possède deux états : vide (*vid*) quand il n' y a aucune pièce sur le dispositif d'arrivée et occupé (*occ*) dès qu'une pièce est arrivée sur ce dernier. Cet événement est décrit dans la clause OPERATIONS par *arr_p*. L'opération *Chgt* exprime la propriété suivante : " la pince ne peut charger que s'il y a une pièce sur *Da* et que si *Dt* est vide ". Il décrit le changement d'état simultané des variables *Da1*(*occ* à *vid*) et *Dt1*(*vid* à *occ*). Nous introduisons aussi deux types de pièces *T1* et *T2* qui devront être évacuées chacune sur leur propre système d'évacuation respectifs *De1* et *De2* sachant que *AUCUN* correspond au fait qu'il n'y a pas de pièce dans la pince. Un capteur permettra de connaître le type de pièce au chargement. Les dispositifs d'évacuation seront gérés par l'opération *evac*.

Le diagramme d'états-transitions de la variable *Type* pour l'opération *Chgt* avant simplification est celui de la figure 8.40 où le prédicat $P$ est faux dans les états *T1* et *T2*. Les transitions issues de ces états disparaissent. Dans l'état *AUCUN* le prédicat $P \widehat{=} (Da1 = occ \wedge Dt1 = vid \wedge Type = AUCUN)$ devient $P \widehat{=} (Da1 = occ \wedge Dt1 = vid)$. Il sera donc noté *IN Da1 = occ* $\wedge$ *Dt1 = vid* entre crochets à la suite du nom de l'opération. La transition réflexive

REFINEMENT *Robot1* REFINES *Robot0*
SETS $TYPE = \{AUCUN, T1, T2\}$
VARIABLES *Dt1*, *Da1*, *De1*, *De2*, *Type*
INVARIANT
$Dt1 = Dt0 \wedge Da1 \in \{vid, occ\} \wedge$
$De1 \in \{vid, occ\} \wedge De2 \in \{vid, occ\} \wedge$
$Type \in TYPE \wedge$
$Type = AUCUN \Rightarrow Dt1 = vid \wedge$
$Type \neq AUCUN \Rightarrow Dt1 = occ$
INITIALISATION
*Dt1*, *Da1*, *De1*, *De2*, *Type* := *vid*, *vid*, *vid*, *vid*, *AUCUN*
OPERATIONS
*Chgt* $\widehat{=}$ ANY *pp* WHERE $pp \in \{T1, T2\} \wedge Da1 = occ \wedge Type = AUCUN$
THEN *Type* := *pp* || *Da1*,*Dt1* : = *vid*, *occ* END
*Dchgt* $\widehat{=}$ SELECT $De1 = vid \wedge Type = T1$
THEN *Dt1*, *De1*, *Type* := *vid*, *occ*, *AUCUN*
WHEN $De2 = vid \wedge Type = T2$
THEN *Dt1*, *De2*, *Type* := *vid*, *occ*, *AUCUN* END
*arr_p* $\widehat{=}$ SELECT $Da1 = vid$ THEN $Da1 := occ$ END
*evac* $\widehat{=}$ SELECT $De1 = occ$ THEN $De1 := vid$
WHEN $De2 = occ$ THEN $De2 := vid$ END
END

FIG. 8.39 – Premier raffinement du robot

autour de l'état *AUCUN* disparaît car elle ne modifie l'état d'aucune variable. On obtient le diagramme de la figure 8.41

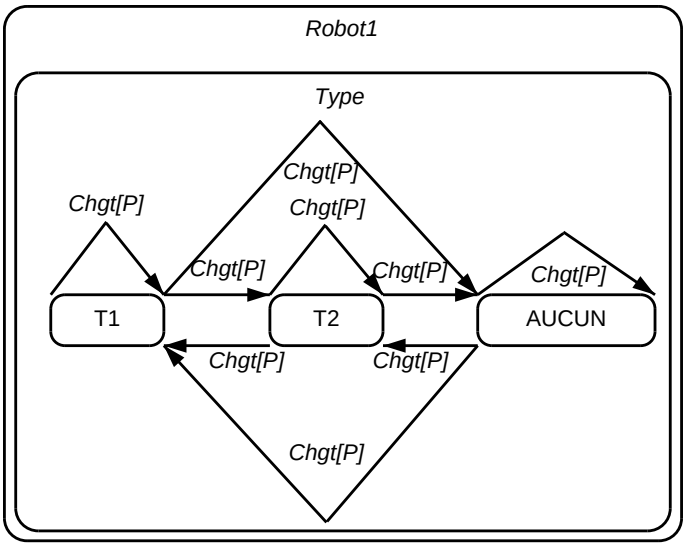


FIG. 8.40 – Diagramme d'états-transitions de la variable *Type* pour *Chgt*

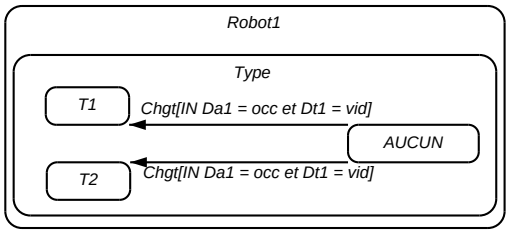


FIG. 8.41 – Diagramme d'états-transitions de la variable *Type* pour *Chgt* après simplification

De la même manière on déduit les diagrammes d'états-transitions pour les autres opérations et l'initialisation. On concatène ces diagrammes pour obtenir le diagramme complet de la variable *Type* (fig 8.42).

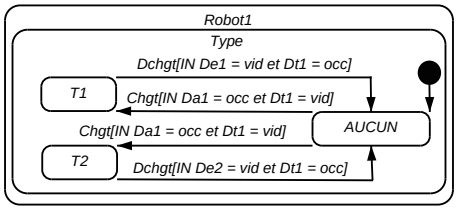
Robot1
Type
Dchgt[IN De1 = vid et Dt1 = occ]
T1
Chgt[IN Da1 = occ et Dt1 = vid]
Chgt[IN Da1 = occ et Dt1 = vid]
AUCUN
T2
Dchgt[IN De2 = vid et Dt1 = occ]

FIG. 8.42 – Diagramme d'états-transitions complet de la variable *Type*

Sur le même principe de déduction on définit les diagrammes d'états-transitions des autres variables de la spécification (fig 8.43) :

- *Dt1*
- *De1*
- *De2*
- *Da1*

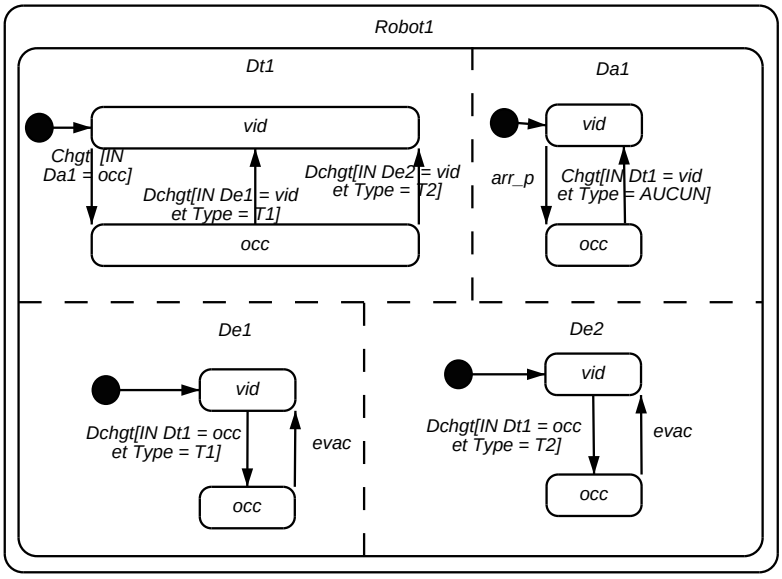


FIG. 8.43 – Diagramme d'états-transitions des variables *Dt1, De1, De2* et *Da1*

Pour obtenir le diagramme complet de la spécification *Robot1* du robot il suffit de calculer le produit synchronisé des sous diagrammes de chaque variable. On commence par le produit synchronisé entre les diagrammes respectifs de *Dt1* et *Type* : $\mathrm{D}_{Dt1,\,Type} = \mathrm{D}_{Dt1} \parallel \mathrm{D}_{Type}/\{$(Chgt, Chgt), (Dchgt, Dchgt)$\}$
Le résultat $\mathrm{D}_{Dt1,\,Type}$ est illustré par la figure 8.44. Les valeurs des variables *Dt1* et *Type* en fonction des états du diagramme sont notées dans le tableau 8.3.

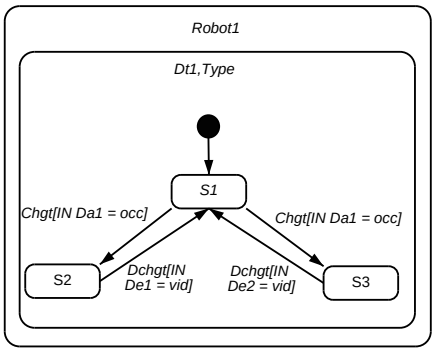


FIG. 8.44 – Diagramme d'états-transitions $D_{Dt1,\,Type}$

| | *Dt1* | *Type* |
|---|---|---|
| *S1* | *vid* | *AUCUN* |
| *S2* | *occ* | *T1* |
| *S3* | *occ* | *T2* |

TAB. 8.3 – Valeurs des variables pour les états de $D_{Dt1,\,Type}$

Ensuite on calcule les diagrammes $\mathrm{D}_{Dt1,\,Type,\,Da1}$, $\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1}$ et $\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1,\,De2}$ par les produits synchronisés :

- $\mathrm{D}_{Dt1,\,Type,\,Da1} = \mathrm{D}_{Dt1,\,Type} \parallel \mathrm{D}_{Da1}/\{$Dchgt, (Chgt, Chgt), arr_p$\}$
- $\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1} = \mathrm{D}_{Dt1,\,Type,\,Da1} \parallel \mathrm{D}_{De1}/\{$(Dchgt, Dchgt), Chgt, arr_p, evac$\}$
- $\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1,\,De2} = \mathrm{D}_{Dt1,\,Type,\,Da1,\,De1} \parallel \mathrm{D}_{De2}/\{$(Dchgt, Dchgt), Chgt, arr_p, evac$\}$

Le résultat de la concaténation des diagrammes de chaque variable ($\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1,\,De2}$) dont les étapes sont les diagrammes $\mathrm{D}_{Dt1,\,Type,\,Da1}$ et $\mathrm{D}_{Dt1,\,Type,\,Da1,\,De1}$ est illustré par la figure 8.45 et les valeurs des variables pour chaque état par le tableau 8.4.

#### 8.11.3.3 Validation des propriétés

On peut maintenant vérifier les propriétés telles que : "*Dt* ne peut charger une pièce que si *Dt* est libre (*vid*) et que *Da* est occupée (*occ*)". Sur le diagramme d'états-transitions le chargement de pièce (opération *Chgt*) s'effectue depuis l'état *S4* et les valeurs respectives de *Dt1* et *Da1* sont respectivement *vid* et *occ*. Le système respecte cette propriété. On pourrait aussi le vérifier pour les autres propriétés.

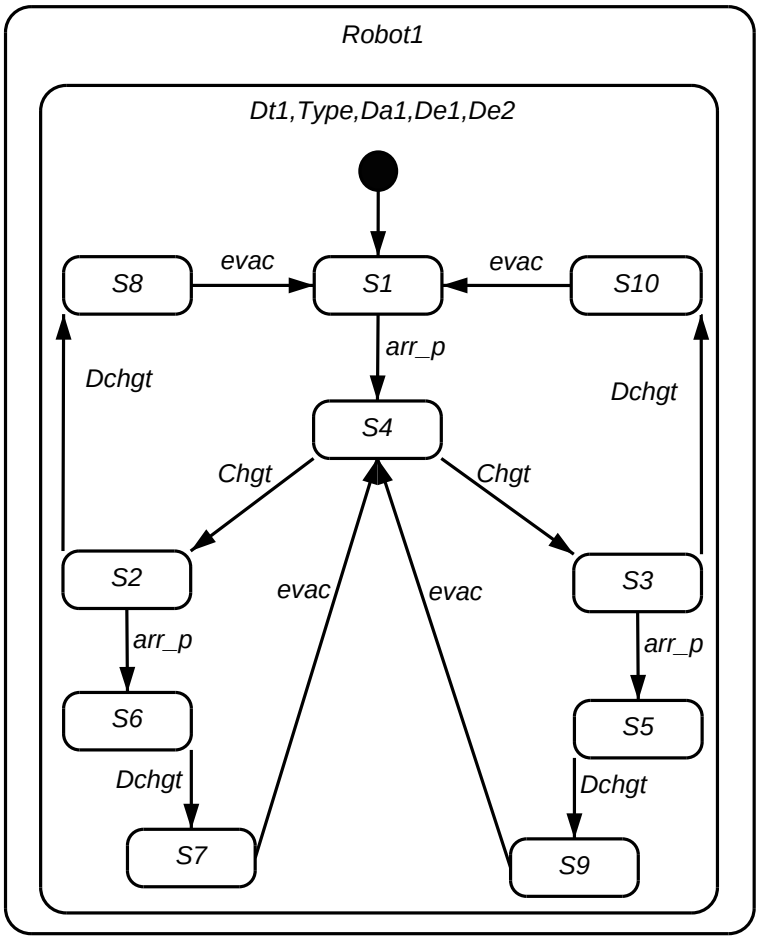


FIG. 8.45 – Diagramme d'états-transitions $D_{Dt1,Type,Da1,De1,De2}$

| | Dt1 | Type | Da1 | De1 | De2 |
|---|---|---|---|---|---|
| S1 | vid | AUCUN | vid | vid | vid |
| S2 | occ | T1 | vid | vid | vid |
| S3 | occ | T2 | vid | vid | vid |
| S4 | vid | AUCUN | occ | vid | vid |
| S5 | occ | T2 | occ | vid | vid |
| S6 | occ | T1 | occ | vid | vid |
| S7 | vid | AUCUN | occ | occ | vid |
| S8 | vid | AUCUN | vid | occ | vid |
| S9 | vid | T2 | occ | vid | occ |
| S10 | vid | T2 | vid | vid | occ |

TAB. 8.4 – Valeurs des variables pour les états de $D_{Dt1,Type,Da1,De1,De2}$

## 8.12 Comparaison avec d'autres méthodes

Notre méthode permet de documenter et mieux comprendre les spécifications B en extrayant des vues UML. En effet l'utilisation de méthodes formelles telles que la méthode

B permet d'obtenir un très bon niveau de confiance dans le développement, mais ne donne aucune garantie de validation de la spécification par rapport au problème posé dans le cahier des charges. La méthode utilisée par Bert, Potet et Stouls dans [33, 16, 34] est une démarche différente. En effet elle permet d'extraire un système de transitions étiquetées complet représentant la totalité des comportements du système quelle que soit la spécification B événementielle considérée. Bert traite tout le langage B y compris les systèmes à nombre infini d'états en les représentant par des diagrammes d'états fini dont les états sont des ensembles définis de manière symbolique. Dans [33], cette méthode s'applique aux raffinements, comme dans l'exemple, contrôle d'accès à un parking en fonction du nombre de places et des voitures déjà en stationnement, illustré par les figures 8.46, 8.47, 8.48 et 8.49, de manière à ajouter des sous-états à l'intérieur des états correspondant aux états du système de transition de l'abstraction qui, par conséquent, deviennent les super-états du raffinement considéré. En effet les trois états de la figure 8.47 correspondant aux valeurs de la variable *cc* sont conservés dans le diagramme extrait du raffinement (fig 8.49). L'état correspondant à la valeur 0 de la variable *cc* devient un super-état incluant deux sous-états correspondant à la valeur de la variable *feu*.

MACHINE *Parking*
CONSTANTS *NbMax*
PROPERTIES $NbMax > 0$
VARIABLES *NbVoit, cc*
INVARIANT
$NbVoit \in 0..NbMax \wedge$
$cc \in -1..1 \wedge$
$(cc = -1 \Rightarrow NbVoit < NbMax) \wedge$
$(cc = 1 \Rightarrow NbVoit > 0)$
INITIALISATION
$NbVoit := 0 \parallel cc := 0$
OPERATIONS
*entrer* $\widehat{=}$ SELECT $NbVoit < NbMax \wedge cc = 0$ THEN $NbVoit := NbVoit + 1 \parallel cc := 1$ END
*controler_entree* $\widehat{=}$ SELECT $cc = 1$ THEN $cc := 0$ END
*sortir* $\widehat{=}$ SELECT $NbVoit > 0 \wedge cc = 0$ THEN $NbVoit := NbVoit - 1 \parallel cc := -1$ END
*controler_sortie* $\widehat{=}$ SELECT $cc = -1$ THEN $cc := 0$ END
END

FIG. 8.46 – La machine *Parking*

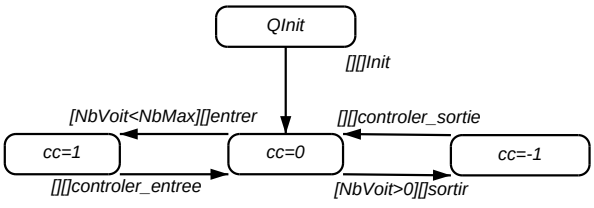


FIG. 8.47 – Système de transitions étiquetées relatif à la machine *Parking*

REFINEMENT *Parking_r1*
SETS *Couleur_feu* = {*vert, rouge*}
VARIABLES *feu*
INVARIANT
*feu* ∈ *Couleur_feu* ∧
*((cc=0* ∧ *feu=vert)* ⇒ *NbVoit* < *NbMax)* ∧
*((cc=0* ∧ *feu=rouge)* ⇒ *NbVoit = NbMax)* ∧
*(cc = 1* ⇒ *feu = vert)* ∧
*((cc=-1* ∧ *feu=rouge)* ⇒ *NbVoit = NbMax - 1)* ∧
*((cc=-1* ∧ *feu=vert)* ⇒ *NbVoit* < *NbMax - 1)*
INITIALISATION
*NbVoit := 0* || *cc := 0* || *feu := vert*
OPERATIONS
*entrer* ≙ SELECT *cc = 0* ∧ *feu = vert* THEN *NbVoit := NbVoit + 1* || *cc := 1* END
*controler_entree* ≙ SELECT *cc = 1* THEN
IF *NbVoit = NbMax* THEN *feu := rouge* END ||
*cc := 0* END
*sortir* ≙ SELECT *NbVoit* > *0* ∧ *cc = 0* THEN *NbVoit := NbVoit - 1* || *cc := -1* END
*controler_sortie* ≙ SELECT *cc = -1* THEN
IF *feu = rouge* THEN *feu := vert* END ||
*cc := 0* END
END

FIG. 8.48 – Le raffinement de la machine *Parking*

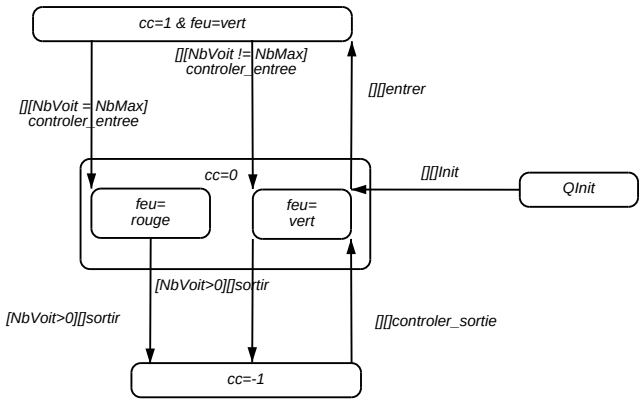


FIG. 8.49 – Système de transitions étiquetées relatif au raffinement *Parking_r1*

Cet aspect de granularité n'a pas été retenu dans nos travaux car notre démarche consiste à définir un diagramme d'états-transitions faisant apparaître une partie des variables pour mettre en relief une partie du comportement du système relatif à un aspect particulier des propriétés complexes de l'invariant, c'est-à-dire une règle de gestion du cahier des charges. La taille des diagrammes d'états-transitions extraits d'une spécification B est une fonction exponentielle du nombre de variables et du nombre de valeur qu'elles peuvent prendre. Moins

on représente de variables plus le diagramme est lisible et exploitable. Dans une optique de concision du diagramme il est donc nécessaire de restreindre au maximum l'ensemble des variables représentées.

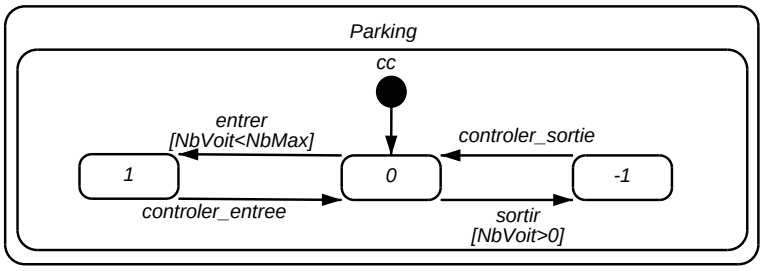


FIG. 8.50 – Diagramme d'états-transitions relatif à la machine abstraite *Parking*

Une autre particularité de cette démarche réside dans la scission des conditions de transition en deux groupes : conditions de déclenchement, première condition entre crochets des transitions de la figure 8.47 (*NbVoit* < *NbMax* et *NbVoit* > *0*) et conditions d'atteignabilité, deuxième condition entre crochets 8.49 (*NbVoit != NbMax* et *NbVoit = NbMax*). Cette technique s'applique à une approche événementielle de la méthode B où la garde est une condition nécessaire pour que l'événement se déclenche. La garde de la substitution principale de l'événement est donc la condition de déclenchabilité de l'événement. Il est possible d'utiliser une autre substitution conditionnelle à l'intérieur de la substitution principale et la condition de cette deuxième substitution est la condition d'atteignabilité de l'état cible de la transition. En d'autres termes, la condition de déclenchabilité correspond au raffinement de la garde de l'événement de la machine abstraite et la condition d'atteignabilité à la différence d'état cible de la transition lorsque des sous-états apparaissent dans un super-état de l'abstraction.

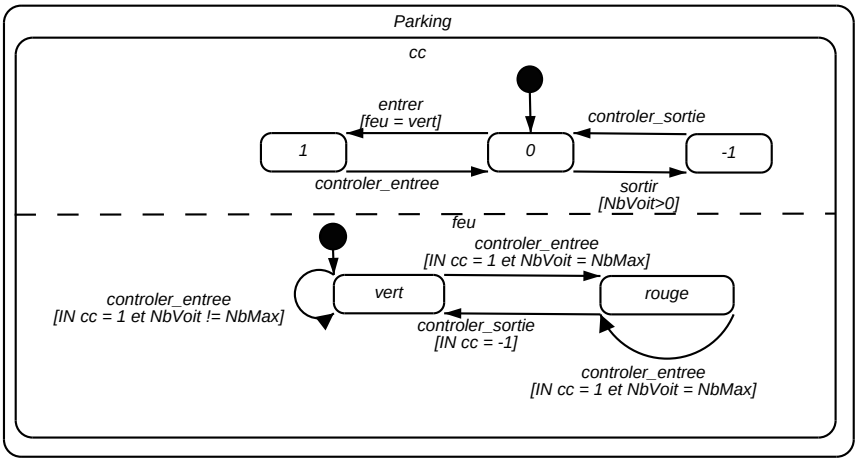


FIG. 8.51 – Diagramme d'états-transitions des variables du raffinement *Parking_r1*

Il existe une différence entre le système de transitions étiquetées obtenu à l'aide de l'approche de Bert, Potet et Stouls (fig 8.49) et le diagramme d'états-transitions complet (fig 8.52) obtenu avec notre approche, illustrée par les figures 8.50, 8.51, 8.5 et 8.52, pour la spécification du raffinement de l'exemple du parking, *Parking_r1*. En effet l'état *cc = -1* de la

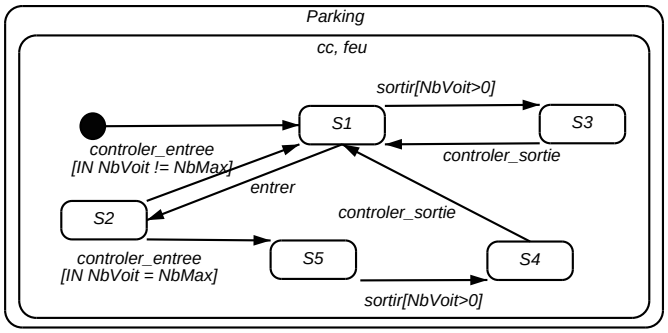


FIG. 8.52 – Diagramme d'états-transitions $D_{cc,feu}$ du raffinement *Parking_r1*

| | *cc* | *feu* |
|---|---|---|
| *S1* | *0* | *vert* |
| *S2* | *1* | *vert* |
| *S3* | *-1* | *vert* |
| *S4* | *-1* | *rouge* |
| *S5* | *0* | *rouge* |

TAB. 8.5 – Valeurs des variables pour les états de $D_{cc,feu}$

figure 8.49 correspond à deux états, *S3* et *S4* de la figure 8.52. Ces divergences proviennent de la manière de construire le graphique. Les systèmes de transitions étiquetées selon Bert, Potet et Stouls sont élaborés à partir d'une disjonction de l'invariant dans la clause ASSERTION alors que les diagrammes d'états-transitions, selon notre démarche, sont construits depuis les diagrammes respectifs des variables de la spécification. Par exemple pour l'outil GeneSyst [34], il est nécessaire d'ajouter la ligne ASSERTION *cc=-1 or cc=0 or cc=1* dans la spécification de la figure 8.46.

# Chapitre 9

# Diagrammes de séquence et d'activités

## 9.1 Introduction

Les précédents chapitres s'appliquaient à extraire des vues UML depuis les spécifications abstraites d'un projet B. Les implémentations d'un projet B sont écrites en B0. Ce sous-ensemble du langage B est assez proche des langages de programmation impératifs traditionnels. Par conséquent la compréhension des implémentations est beaucoup plus aisée que celle des spécifications abstraites. Néanmoins il serait intéressant d'extraire des spécifications concrètes des éléments permettant une meilleure compréhension globale de leur fonctionnement. Dans cette optique nous avons choisi d'en extraire deux types de diagrammes qui sont :

- les diagrammes de séquence,
- les diagrammes d'activités.

## 9.2 Diagramme de séquence

On peut traduire une spécification B en un diagramme de séquence si les différentes machines peuvent communiquer. Le niveau où cette communication est la plus présente est l'implémentation. Elle est réalisée par les appels successifs des opérations.

### 9.2.1 Les objets

On définit les objets par rapport aux clauses IMPORTS et SEES. Dans tous les cas, les machines importées par la clause IMPORTS deviennent des objets du diagramme de séquence. Les machines vues par la clause SEES deviennent des objets uniquement dans le cas où ses opérations sont utilisées. Ceci est possible uniquement dans le cas où les opérations de la machine ne modifient pas les variables de la machine. D'autre part chaque instance d'une même machine doit être représentée dans le diagramme de séquence. Si une machine est importée plusieurs fois et renommée (fig 9.1) elle apparaîtra autant de fois qu'elle est importée dans le diagramme de séquences.

Chaque objet est représenté par un rectangle et une ligne de vie dans le diagramme de séquence (fig 9.2). La ligne verticale correspond à l'écoulement du temps.

IMPLEMENTATION *MAIN_imp*
IMPORTS *M1.M,M2.M*
...

FIG. 9.1 – Une implémentation

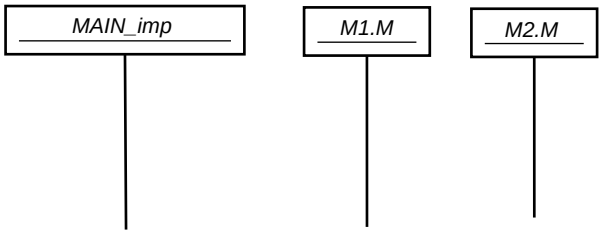


FIG. 9.2 – La traduction d'une implémentation en UML

### 9.2.2 Les messages

Chaque appel d'opération permet la communication entre les différents composants d'un projet B. Au niveau de l'IMPLEMENTATION, les exécutions des différentes instructions sont séquentielles. On entame l'exécution de l'instruction *n* une fois que l'exécution de l'instruction *n-1* est terminée. Dans la figure 9.3, l'exécution du système débute par l'appel de l'opération *op1* de *M1*. Par la suite l'opération *op2* de *M2* est exécutée.

IMPLEMENTATION *MAIN_imp*
IMPORTS *M1.M,M2.M*
...
OPERATIONS
*main*$\widehat{=}$BEGIN *M1.op1 ;M2.op2* END
END

FIG. 9.3 – Une opération en B0

Au niveau du diagramme de séquence (fig 9.4), l'appel des opérations M1.op1 et M2.op2 est représenté par un message partant de la ligne du temps de l'objet MAIN (objet dont l'opération est appelante [main]) vers l'objet dont l'opération est appelée M1.M pour M1.op1 et M2.M pour M2.op2. Le message est libellé par le nom de l'opération appelée. Le temps d'exécution des méthodes de l'objet se matérialise par les rectangles sur la ligne d'écoulement du temps. Ces rectangles symbolisent le temps d'exécution de l'opération. Le temps d'exécution de l'opération appelante main englobe les temps des opérations appelées M1.op1 et M2.op2. Le retour de l'exécution sur l'opération main est implicite. D'autre part l'appel de l'opération M2.op2 s'effectue une fois que le temps d'exécution de l'appel de M1.op1 est écoulé car on ne peut pas exécuter deux opérations simultanément en B0.

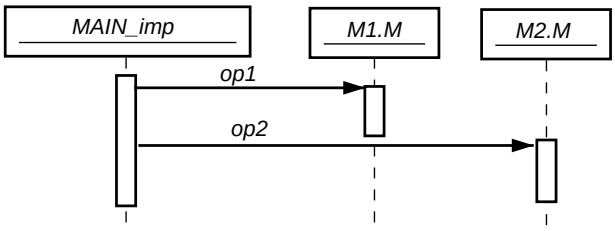


FIG. 9.4 – La traduction de l'appel d'une opération

### 9.2.3 Les paramètres et valeurs de retour des opérations

Dans le langage B, les opérations peuvent retourner une ou plusieurs valeurs. C'est le cas de l'exemple suivant où l'opération *M1.op1* retourne deux valeurs (fig 9.5).

IMPLEMENTATION *MAIN_imp*
IMPORTS *M1.M*
...
OPERATIONS
*main*$\widehat{=}$VAR *aa,bb* IN *aa,bb*$\leftarrow$*M1.op1* END
END

FIG. 9.5 – Une opération qui retourne des valeurs

Dans le diagramme de séquence correspondant (fig 9.6) on affiche explicitement le retour à l'exécution de l'opération principale. Ce message est libellé par le nom de la variable qui va être affectée par la valeur retournée par l'opération.

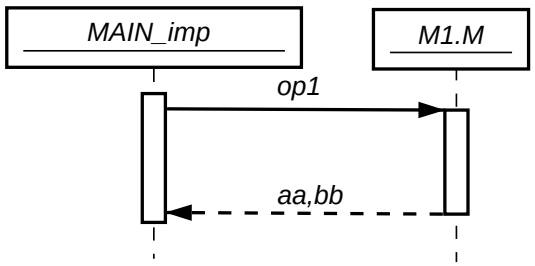


FIG. 9.6 – Traduction d'une opération qui retourne des valeurs

| Instruction IF | Instruction CASE |
|---|---|
| IMPLEMENTATION *MAIN_IMP* | IMPLEMENTATION *MAIN_IMP* |
| IMPORTS *M1.MX* | IMPORTS *M1.MX* |
| ... | ... |
| OPERATIONS | OPERATIONS |
| *main*$\widehat{=}$ BEGIN | *main*$\widehat{=}$ BEGIN |
| ... | ... |
| IF *v=Value1* THEN *M1.op1* | CASE *v* OF |
| ELSIF *v=Value2* THEN *M1.op2* | EITHER *Value1* THEN *M1.op1* |
| ELSE *M1.op3* | OR *Value2* THEN *M1.op2* |
| END | ELSE *M1.op3* |
| END | END |
| | END |

FIG. 9.7 – Instructions IF et CASE

### 9.2.4 Les conditions

Le niveau IMPLEMENTATION du langage B possède des instructions conditionnelles. Ces instructions sont IF et CASE (tableau 9.7). Dans les spécifications suivantes la branche ELSE a pour condition la négation des conditions des branches qui la précèdent.

Dans le diagramme de séquence correspondant aux deux spécifications définies dans le tableau précédent les conditions utilisées par les instructions IF et CASE sont les mêmes donc on obtient le même diagramme de séquence. Les messages conditionnels ont la même origine et se distinguent par les conditions de gardes placées devant le message. On dédouble la ligne de vie de l'objet destinataire du message car les trois possibilités peuvent apparaître selon la valeur de la variable $v$ à la même date. Dans la spécification aucune valeur n'est retournée par les opérations donc on ne représente pas explicitement le retour à l'exécution de l'opération appelante.

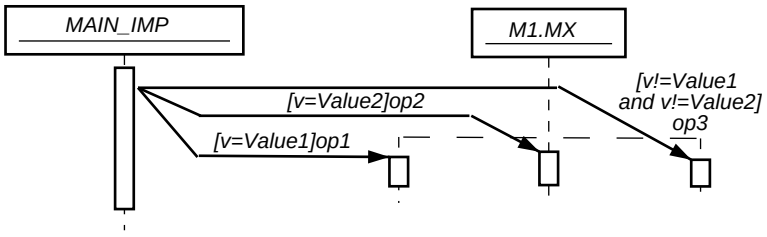


FIG. 9.8 – Traduction des appels d'opération à l'intérieur d'une instruction conditionnelle

### 9.2.5 Les boucles dans les diagrammes de séquence

En B l'instruction WHILE permet de créer des boucles (fig 9.9). On ne peut pas traduire cette instruction particulière comme les autres.

Dans le diagramme de séquence correspondant on introduit du pseudo-code pour définir la boucle à côté de la ligne de vie des objets. En introduisant du pseudo-code on peut donc délimiter la boucle; il ne reste plus qu'à insérer les message envoyés à l'intérieur du corps de

```
IMPLEMENTATION MAIN_imp
IMPORTS M1.MX
...
OPERATIONS
    main≙VAR fin IN
        fin := 25
        WHILE fin > 0 DO
            M1.op1 ;
            fin := fin - 1 ;
        INVARIANT fin ∈ NAT
        VARIANT fin END
        END
    END
END
```

FIG. 9.9 – Une boucle en B0

la boucle dans le diagramme de séquence entre les délimiteurs de début et de fin de boucle (fig 9.10).

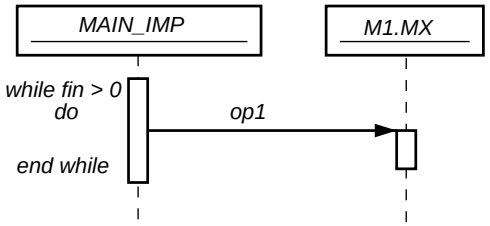


FIG. 9.10 – Traduction d'une boucle

## 9.3 Diagrammes d'activités

Les machines abstraites utilisent le langage B des substitutions généralisées à la différence des implémentations qui utilisent le langage B0 composé d'instructions. Les opérations définies dans les implémentations sont les raffinements des opérations des machines abstraites. Elles possèdent les mêmes résultats et paramètres. Elles utilisent des données concrètes au lieu des données abstraites utilisées dans les machines abstraites. Nous allons montrer une traduction d'un sous-ensemble représentatif de ces instructions en diagramme d'activités.

### 9.3.1 Les acteurs du diagramme d'activités

En B on peut appeler les opérations/services des machines abstraites instanciées avec la clause IMPORTS. Dans le cas où l'opération ne modifie pas les variables de la machine on peut aussi appeler les opérations des machines vues par la clause SEES (fig 9.11).

IMPLEMENTATION *MAIN_imp*
IMPORTS *aa.M1,bb.M1*
SEES *M2*
...
END

FIG. 9.11 – Liens de visibilité et importation entre machines

Dans un premier temps on crée une travée pour la machine *MAIN* qui représente le point d'entrée du diagramme d'activité. Ensuite toutes les instances de machines apparaissent sous la forme de travées comme les instances de *M1* : *aa* et *bb* dans la figure 9.12. Les machines vues apparaissent uniquement si les opérations de ces machines sont utilisées. Il faut aussi donner le type de l'instance ; pour *aa* et *bb* le type est *M1*. Pour *M2* le nom de l'instance est le type. A travers cette traduction on connaît les opérations des instances.

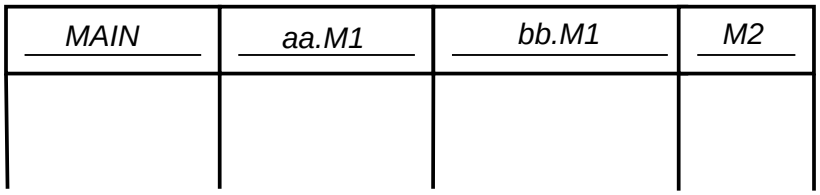


FIG. 9.12 – Traduction des liens de visibilité et importation entre machines

### 9.3.2 Initialisation, opération et états d'acceptation

L'initialisation permet de donner une valeur initiale à chaque variable de la machine abstraite, ensuite les opérations utilisent ces variables (fig 9.13). L'opération se termine une fois que toutes ces instructions sont exécutées.

IMPLEMENTATION *MAIN_imp*
...
INITIALISATION ...
OPERATIONS
*OP1*$\widehat{=}$
...
END

FIG. 9.13 – Initialisation et opérations dans une machine B

Dans le diagramme d'activités correspondant, l'initialisation est symbolisée par un disque plein et une flèche qui est dirigée vers le premier état de la clause INITIALISATION. Le dernier état extrait de l'opération traduite possède une transition dirigée vers l'état final symbolisé par un disque plein entouré d'un cercle (fig 9.14).

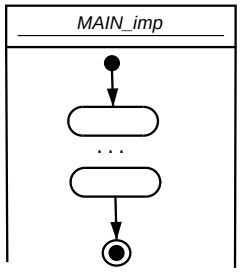


FIG. 9.14 – Traduction de l'initialisation et des opérations

### 9.3.3 Séquentialité

Le séparateur d'instruction utilisé dans B0 est le caractère ';'. Il exprime la séquentialité des instructions (fig 9.15). La séquentialité intervient autant dans l'initialisation que dans les opérations.

IMPLEMENTATION *MAIN_IMP*
REFINES *MAIN*
INITIALISATION *Inst_ini_1 ;... ;Inst_ini_n*
OPERATIONS
*main*$\widehat{=}$BEGIN *Inst1 ;... ;Instn* END
END

FIG. 9.15 – Le séquencement

Dans le diagramme d'activités correspondant (fig 9.16) on retrouve l'initialisation et l'état d'acceptation décrits dans le paragraphe précédent. Chaque instruction est transformée en une transition où l'instruction apparaît précédé d'un caractère '/'. Ceci signifie dans le formalisme des diagrammes d'activités que l'instruction est exécutée durant la transition. Le diagramme possède deux séquences de transitions. La première séquence représente les instructions *Inst_ini_1* à *Inst_ini_n* de la clause INITIALISATION. La deuxième séquence représente les instructions *Inst1* à *Instn* de l'opération *OP1*.

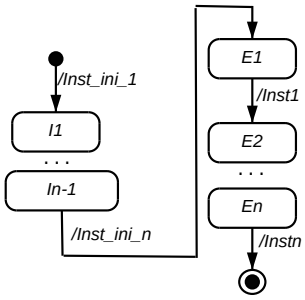


FIG. 9.16 – Traduction du séquencement

### 9.3.4 Expressions conditionnelles

#### 9.3.4.1 Instructions IF et CASE

Les instructions conditionnelles sont souvent écrites avec une instruction IF ... THEN ... ELSE ... comme dans l'exemple ci-dessous (fig 9.17).

IF *aa < MAX THEN Inst1*
ELSE *Inst2* END

FIG. 9.17 – Instructions conditionnelles simples

Le diagramme UML correspondant à ce type d'instruction (fig 9.18) possède un losange symbolisant le point de choix. Les deux transitions issues du losange permettent d'indiquer la condition et l'instruction associée.

Il permet de connaître la condition entre crochet suivi, comme il est défini dans le paragraphe précédent, du caractère '/' et de l'instruction correspondant à la condition.

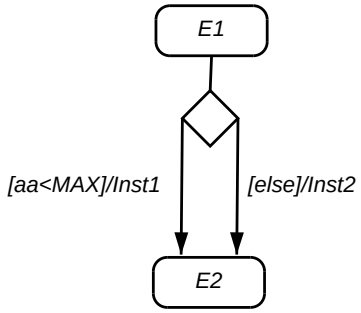


FIG. 9.18 – Traduction des expressions conditionnelles simples en UML

L'instruction IF peut apparaître sous plusieurs formes. Une deuxième forme est proche de l'instruction CASE (fig 9.19). Dans ce cas, on a un point de choix avec n possibilités de transition et n > 2.

La transformation en diagramme d'activités de ces deux instructions est différente :

- IF : on insère directement la condition entre crochets
- CASE : on met comme condition entre crochets *var=valuex* où *x* est défini par l'ensemble {1,2...n}.

Une fois que l'on a défini la condition, on insère l'instruction correspondant à la condition à la suite du caractère '/'. Dans le cas où il y a plusieurs instructions à la suite dans la même branche de l'expression conditionnelle on crée une séquence d'états avec une transition par instruction.

#### 9.3.4.2 L'expression BOOL

L'expression BOOL est écrite à l'aide d'une expression conditionnelle. Elle renvoie TRUE ou FALSE selon que l'expression entre parenthèses est vraie ou fausse (fig 9.21).

Cette expression (BOOL(*Cond* )) peut être traduite par l'utilisation de l'instruction IF *Cond* THEN *aa* := TRUE ELSE *aa* := FALSE END. Dans le diagramme d'activité correspondant la

(
IF *Cond1* THEN *Inst1*
ELSEIF *Cond2* THEN *Inst2*
...
ELSE *Instn* END
)

|

(
CASE *var* OF
EITHER *value1* THEN *Inst1*
OR *value2* THEN *Inst2*
...
ELSE *Instn* END
)

FIG. 9.19 – Expressions conditionnelles complexes

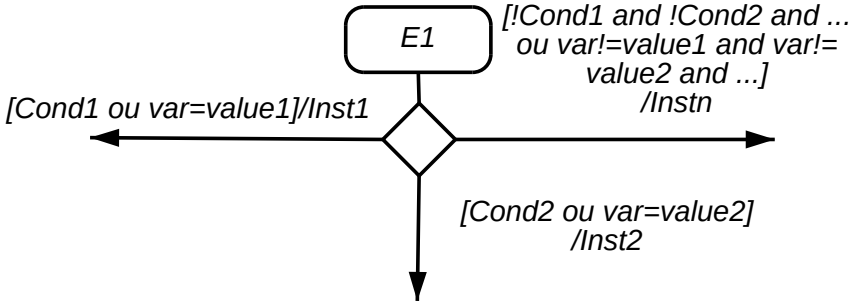


FIG. 9.20 – Traduction des expressions conditionnelles complexes en UML

*aa* := BOOL(*bb* = *xx*)

FIG. 9.21 – Expressions booléennes

traduction est donc similaire à celle définie précédemment (fig 9.18).

### 9.3.5 Les boucles

L'instruction WHILE permet d'écrire des boucles en langage B0 (fig 9.22). Les mots INVARIANT et VARIANT ne sont pas des instructions mais sont nécessaires pour la preuve en B. Par conséquent elles n'apparaîtront pas dans le diagramme d'activité. Les instructions qui seront traduites sont {Inst_1, ..., Inst_n, Inst_n+1}.

Dans le diagramme d'états-transitions correspondant à la traduction de l'instruction

```
INITIALISATION Inst_0
OPERATIONS
main≙
    VAR ii IN
        ii := 20;
        WHILE ii /= 0 DO
            Inst_1;
            ...
            Inst_n
        INVARIANT ii : NAT
        VARIANT ii
        END;
        Inst_n+1
    END
```

FIG. 9.22 – Une boucle en B0

WHILE (fig 9.23) on observe une boucle générée par les transitions et états issus de la traduction des instructions appartenant au corps de la boucle WHILE. La condition en fonction de la variable *ii* permet d'exécuter les instructions internes à la boucle ou d'en sortir ce qui est visible sur le schéma. La sortie de la boucle passe par l'état *En+1*. La branche du diagramme qui correspond à la boucle est constituée des états *E2,...,En*.

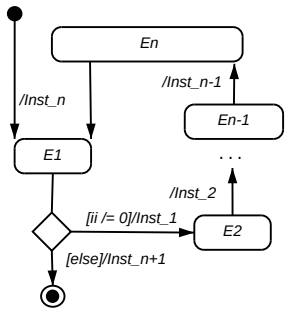


FIG. 9.23 – Traduction d'une boucle

### 9.3.6 Appel d'opérations, paramètres et valeurs de retour

En B opérationnel les spécifications possèdent des clauses de composition. Elles importent d'autres machines, appartenant au même projet B, dans le but d'en utiliser les services (opérations) (fig 9.24). Il existe une synchronisation entre la machine principale, point d'entrée de l'exécution du projet, et les autres machines dont les opérations sont appelées. Dans la spécification ci-dessous la machine *M* est importée sous le nom *MX*. Elle est ensuite appelée par ce nom pour en utiliser les opérations. Les opérations peuvent prendre des arguments et retourner des valeurs à la fin de leur exécution (fig 9.24). La valeur des arguments est fixée avant l'exécution de la première instruction de l'opération appelée.

```
IMPLEMENTATION MAIN_IMP
...
IMPORTS MX.M
...
OPERATIONS
main≙
        VAR zz IN
                zz←MX.OP1(25);
        END
END


IMPLEMENTATION M_IMP
OPERATIONS
        res←OP1(arg1) defs BEGIN ... END;
...
END;
```

FIG. 9.24 – Appel d'opération en B

Le diagramme d'activités extrait de ces spécifications (fig 9.24) fait apparaître l'appel de l'opération au niveau de la transition (fig 9.25) qui part de la travée *MAIN* vers la travée *MX*. L'appel d'opération est noté par le caractère '^'. Le retour de la séquence d'exécution dans la machine *MAIN* est réalisé par une transition partant de l'état correspondant à l'exécution de l'opération dans la travée *MX* et arrivant dans la travée *MAIN*.

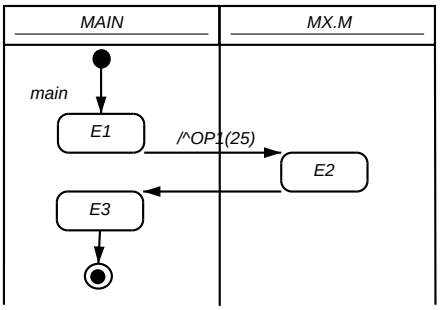


FIG. 9.25 – Traduction de l'appel d'opération

Plus généralement, les diagrammes d'activités sont capables de montrer à peu de choses près les mêmes aspects de communication schématisés par les diagrammes de séquence présentés en début de chapitre. En effet les diagrammes de séquence mettent en avant les messages circulant entre les objets. Dans notre traduction nous avons utilisé ces mécanismes pour symboliser les appels de méthodes d'un objet depuis l'exécution d'une méthode d'un autre objet. Dans les diagrammes d'activités, le symbolisme n'est pas le même, les objets sont représentés par des travées, et la communication correspond aux changement de travées.

Remarque : Un exemple est disponible dans le chapitre 16 dans les figures 16.5, 16.6, 16.16

et 16.17.

# Chapitre 10

# Conclusion sur l'extraction de vues graphiques

Notre contribution dans l'intégration des méthodes formelles et semi-formelles est réalisé par l'extraction de vues UML et de contraintes OCL depuis une spécification B. Cette extraction s'effectue selon deux directions. La première correspond à l'extraction des informations statiques de la spécification et l'expression de :

- Diagrammes de classes,
- Contraintes OCL.

La deuxième permet de montrer le comportement du système ; ceci correspond aux informations dynamiques de la spécification exprimées à l'aide des diagrammes suivant :

- Diagrammes d'états-transitions,
- Diagrammes d'activités,
- Diagrammes de séquence.

## 10.1 Extraction des informations statiques

### 10.1.1 Extraction des diagrammes de classes

Notre démarche consiste à extraire le diagramme de classes représentatif d'une spécification B. Ceci est réalisé par rapport à quatre grands principes :

- les ensembles de typage,
- les contraintes OCL associées aux ensembles de typage,
- les liens entre machines,
- les variables scalaires ou ensemblistes.

Le principal inconvénient de notre méthode par rapport à celle de Fekih [32] est l'utilisation de contraintes OCL pour différencier les fonctions, bijections, injections et surjections totales ou partielles. Néanmoins l'utilisation de stéréotypes pallie ce manque de précision au niveau purement graphique. En outre les avantages de notre démarche sont très importants. Elle est uniforme contrairement à la méthode de Fekih en ce qui concerne les ensembles. Il est donc possible d'automatiser le processus d'extraction puisqu'il n'est pas nécessaire d'interpréter la spécification B.

### 10.1.2 Génération des contraintes OCL depuis les propriétés complexes de l'invariant

Sur l'échantillon d'exemples que nous avons traité, le passage d'une spécification B à une modélisation UML-OCL s'est avéré généralement possible. Les problèmes pour les propriétés B sont localisés à des déclarations implicites ou à des expressions complexes créant de nouveaux ensembles de couples (relations, fonctions) que nous n'avions pas prévu initialement de traiter dans notre approche de génération du diagramme de classes. Ces problèmes se répercutent aussi pour les opérations lors du passage de la post-condition exprimée en B à la post-condition exprimée en OCL. Les solutions consistent d'une part à (re)structurer la spécification B pour expliciter toutes les déclarations nécessaires et d'autre part à générer les classes et les associations supplémentaires.

Hormis ces problèmes, les contraintes OCL générées nous paraissent lisibles et compréhensibles quoique moins concises que les prédicats exprimés en B. Cela est dû partiellement aux opérateurs disponibles en B qui ne possèdent pas un équivalent strict et direct en OCL mais qui s'expriment par l'association de plusieurs constructions.

### 10.1.3 Synthèse de l'extraction des données statiques

Les diagrammes de classes apportent une vue sur les structures de données utilisées dans la spécification B. Bien qu'ils puissent exprimer certaines contraintes de l'invariant, notamment par les stéréotypes, le typage des variables, etc., ils ne peuvent pas en décrire la totalité. Il était donc nécessaire d'utiliser un autre procédé comme la génération des contraintes OCL pour pallier ces manques. Hormis quelques problèmes de concision, les contraintes OCL nous paraissent lisibles et compréhensibles.

## 10.2 Extraction des informations dynamiques

### 10.2.1 Extraction des diagrammes d'états-transitions

Nous avons présenté l'extraction des diagrammes d'états-transitions depuis une spécification B. Elle traite un cas particulier des spécifications B réalisées autour de spécifications à nombre d'états fini. Son utilisation nécessite une méthode d'écriture particulière des spécifications. Elle permet l'utilisation du langage B restreint à un de ses sous-ensembles autour de données scalaires typées à l'aide des ensembles énumérés. Néanmoins cette méthode permet de vérifier les règles de gestion du cahier des charges. En effet, elle décrit la génération de diagrammes d'états-transitions. Ces diagrammes sont extraits respectivement pour une seule variable mais ils peuvent être fusionnés afin d'obtenir un seul diagramme pour la totalité de la spécification. On peut alors plus facilement vérifier des propriétés invariantes (règle de gestion du cahier des charges).

### 10.2.2 Extraction des diagrammes de séquence et d'activités

Nous avons présenté une extraction des diagrammes de séquence et d'activités depuis une spécification B. Les diagrammes de séquence montrent les interactions entre les différents objets du diagramme de classes. Ceci en termes de temps d'exécution, de datation et d'ordre de lancement des messages de communication. Néanmoins il existe une restriction à tous ces apports qui réside dans le fait que le temps n'est pas formalisable intégralement en B ce qui

limite les apports temporels de l'extraction des diagrammes de séquence. Ceci n'est pas un problème car on n'appauvrit pas la spécification B à travers l'extraction. Les diagrammes d'activités montrent surtout l'exécution interne des opérations dans le détail. De ce fait il fallait utiliser les implémentations des composants B (en langage B0) car les spécifications abstraites ne possèdent pas le niveau de détails adéquat pour ce type d'extraction. Nous remarquons aussi le recouvrement par l'extraction des appels d'opération des aspects de communication entre les objets schématisés par les diagrammes de séquence.

### 10.2.3 Synthèse de l'extraction des informations dynamiques

Les diagrammes d'états-transitions permettent d'exprimer le comportement de manière conceptuelle et plutôt abstrait. Au contraire, les diagrammes de séquence et d'activités montrent une vue détaillée du comportement du système. Il existe un recouvrement de l'extraction des diagrammes de séquence par l'extraction des diagrammes d'activités. L'extraction de l'appel d'opération du diagramme d'activités reprend les principes des diagrammes de séquence. Les diagrammes d'activités sont une variante des diagrammes d'états-transitions dans un registre plus concret.

# Troisième partie

# Génération de code source pour des langages à objets

# Chapitre 11

# Génération de code à partir de spécifications formelles

## 11.1 Introduction

Plusieurs travaux ont déjà abordé la traduction de spécifications formelles vers les langages impératifs traditionnels. On peut regrouper les traductions en deux groupes par rapport aux aspects de modularité. Le premier groupe ne conserve pas les aspects de modularité comme les traductions :

- De B vers C [8],
- De B vers OCaml [79].

L'autre groupe maintient la notion de modularité ou la substitue par la notion objet de classes comme les travaux suivants :

- De ObjectZ vers Java[57]/C++[41],
- De VDM++ vers Java/C++ [52, 53, 54, 55].

Nous allons commencer par exposer les objectifs et par donner un aperçu des travaux du projet RNTL[1] BOM[2]. Ensuite nous présenterons un état de l'art général à propos des différentes traductions depuis une spécification formelle vers un langage impératif avant que les outils de traduction du projet BOM soient réalisés.

## 11.2 Projet BOM

Cette partie de notre travail, la génération de code, a été réalisée dans le cadre du projet BOM (B Optimisé Mémoire). Ce projet vise à traduire des spécifications B en langage compilable et exécutable de manière optimisé notamment en terme d'occupation mémoire. L'objectif final est de produire, à partir de spécifications B, des programmes pour des plate-formes embarquées par exemple des cartes à puce. En 2000, date du début du projet BOM, la société Gemplus initiateur du projet BOM utilisait l'Atelier B de ClearSy mais le code C obtenu à partir de spécifications B ne convenait pas pour des applications cartes à puce car

[1] Réseau National des Technologies Logicielles : le Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL) a été mis en place, dès 1999, par le ministère de la Recherche et le ministère de l'Industrie. Il a reçu comme mission de favoriser la constitution de projets innovant de recherche et de développement coopératif entre entreprises et équipes de la recherche publique.

[2] De septembre 2001 à mars 2003 : http ://lifc.univ-fcomte.fr/∼tatibouet/WEBBOM

il était trop gourmand en ressources mémoires. Nous allons exposer les objectifs du projet BOM et poursuivre par un état de l'art relatif aux travaux entrepris dans ce dernier.

## 11.3 Objectifs du projet BOM

Les cartes à puce sont un moyen de sécuriser un système d'information. Les cartes traditionnelles souffraient de leur manque de flexibilité et de leur difficulté de programmation. Afin de pallier ces limitations, un nouveau type de cartes a été conçu. Ces cartes dites ouvertes ne sont plus réservées aux spécialistes car elles acceptent des langages de haut niveau tel que Java. De plus, elles permettent de charger plusieurs applications et proposent à ces applications des mécanismes de coopération. Jusqu'à présent les méthodes de développement classiques ont permis de développer des applications carte ayant un haut niveau de qualité cependant l'émergence du code mobile et la possibilité de charger du code après la mise en service introduit de nouveaux besoins en terme de méthodes de développement et de qualification des cartes à puce par rapport aux problématiques de sécurité et de mobilité du code. Dans ce domaine les méthodes formelles semblent très prometteuses malheureusement les méthodes et outils actuellement disponibles (B, Coq, Isabelle, ...) ne permettaient pas de générer du code de manière efficace répondant aux contraintes très sévères de la carte à puce.

Certes il existait des traducteurs de B vers C/C++ ou ADA mais par exemple la traduction vers C, langage très utilisé pour les cartes à puces, produisait un code permettant l'instanciation de machines mais restant complexe et gourmand en mémoire (utilisation de paramètres pointeur sur les structures correspondant aux machines instanciées dans les fonctions traduites des opérations par le type pointeur : `T_BWallet *` - fig 11.1) donc peu adapté pour des plate-formes embarquées à fortes contraintes mémoire comme les cartes à puce (la capacité mémoire d'une carte à puce excède rarement les quatre kilos-octets de RAM, trente deux kilos-octets de ROM et d'EEPROM). De plus les traducteurs étaient jugés par les ingénieurs de ClearSy complexes et par conséquent difficiles à maintenir et à déboguer. Gemplus à partir de spécifications B réalisait une génération de code pour cartes à puce manuellement. Celle-ci introduisait une rupture dans le processus de preuve et ne rentabilisait pas l'effort de formalisation par réduction du test unitaire.

Le projet BOM s'est intéressé à améliorer le code généré en terme d'optimisation mémoire car les cartes n'ont que peu de ressources mémoires. Les études ont portées sur la méthode B et le processus de traduction.

## 11.4 Objectif et contribution de mes travaux de thèse

Les travaux dans le cadre du projet BOM (fig 11.2) ont fait émerger la nécessité de deux traducteurs car les caractéristiques des langages C d'un côté et Java de l'autre sont très différentes, et les applications visées à travers ces langages aussi :

- un pour le langage C [8] avec la gestion fine des entiers, afin de générer la partie système : Système d'exploitation (JCRE : Java Card Runtime Environment) et Machine virtuelle Java (JCVM : Java Card Virtual Machine),
- un pour le langage Java destiné à la partie applicative : Applets JavaCard exécutées par la machine virtuelle Java.

Notre travail s'est concentré sur l'étude, la conception et la réalisation d'une souche de traducteurs pour les langages à objets qui intègre les études faites dans le cadre du projet

```
#include <stdlib.h>
#include "BWallet.h"

T_BWallet *function new_T_BWallet( void){
    T_BWallet *_this = (T_BWallet *)malloc( sizeof(T_BWallet));
    memset(_this, 0, sizeof(T_BWallet) );
    return _this;
}

//--------------------
// INITIALISATION
//--------------------

//IMPORTSs imported instances
void function T_BWallet_IMPORTS(T_BWallet *_this) {
    _this->initialisation = FALSE;
} // end T_BWallet : :IMPORTS

void function T_BWallet_INITIALISATION(T_BWallet *_this) {
    if (_this->initialisation == FALSE) {
        _this->initialisation = TRUE;
        _this->balance = 0;
    }
} //end T_BWallet : :INITIALISATION;

//access methods
T_int function T_BWallet_get_balance(T_BWallet *_this) {
    return _this->balance;
};

//-----------------
//Clause OPERATIONS
//-----------------

void function T_BWallet_setBalance(T_BWallet *_this,T_int balanceInit) {
    _this->balance = balanceInit;
}

void function T_BWallet_debit(T_BWallet *_this,T_int debitAmount) {
    _this->balance = _this->balance - debitAmount;
}
...
```

FIG. 11.1 – Exemple du *BWallet* (Spécification B : fig 13.2) traduit à l'aide du traducteur C de l'AtelierB

BOM (passage de paramètres par référence, instanciation ou pas des machines, typage fin des entiers, ...) pour économiser la mémoire. Les traducteurs réalisés (Java et C#) visent les nouvelles plate-formes technologiques (Java et Dot Net) et leurs déclinaisons pour les applications embarquées.

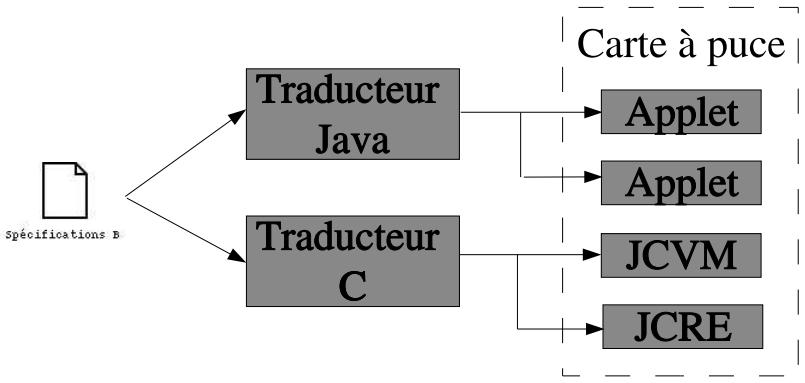


FIG. 11.2 – Objectifs du projet BOM

## 11.5 Etat de l'art

### 11.5.1 Projet BOM : de B vers C

Le traducteur de B0 (sous-ensemble d'implémentation du langage B) vers C utilise la stratégie suivante [8] : un module est une chaîne de développement débutant par une machine abstraite et se terminant par une implémentation. Toutes les machines utilisées (clause SEES et IMPORTS) par les composants de la chaîne en font aussi partie. Le nom de la machine abstraite racine de la chaîne devient le nom du module. Le module aplati est obtenu par concaténation de toutes les données concrètes de la chaîne de développement c'est-à-dire celles de la machine abstraite, des raffinements et de l'implémentation. Ces données concrètes contiennent les ensembles concrets, les constantes concrètes, les variables concrètes, l'initialisation ainsi que les opérations de l'implémentation. Les types retenus pour chaque donnée sont ceux définis en B0. Pour éviter les collisions de nom entre les différentes données des différentes machines (deux données ayant le même nom dans deux composants de chaînes différentes ne posent pas de problème mais ils sont concaténés dans la même entité), chaque donnée est préfixée par le nom du module où elle apparaît [28]. Ceci permet de travailler avec un seul espace de nom pour le projet entier. Les identifiants renommés sont conservés tels quels durant la traduction en C.

### 11.5.2 De B vers OCaml

Selon Petit [79], la structure de la spécification qui est modulaire en B n'est pas conservée en OCaml car Petit a utilisé pour la traduction le système de modules Harper-Lillibridge-Leroy (que nous appelerons HLL dans la suite du document) présenté dans [66]. Le système HLL a pour objectif de fournir une implantation d'un système de modules de type SML paramétrée

par un langage de base et par son vérificateur de types. Le terme *langage de base* désigne un langage quelconque dépourvu de toutes constructions de modularité. Ce n'est pas le cas de B mais il est possible d'aplatir les spécifications d'un projet entier (plusieurs composants B) par un algorithme de dépliage pour obtenir une seule et unique spécification et c'est ce qui a été réalisé dans ce traducteur. Le reste de la traduction concerne les trois parties suivantes :

- l'état de la spécification (variables et constantes),
- les types manipulés,
- l'utilisation d'assertions,
- les structures de contrôle.

#### 11.5.2.1 Structures de contrôle

Parmi ces constituants, la partie la plus facile est la traduction des structures de contrôle des spécifications B. Ces structures correspondent à un sous-ensemble du langage B0 de B. B0 est un noyau qui habituellement se trouve dans les langages impératifs tel que C, ADA, PASCAL, ... Par conséquent cette traduction ne pose pas de problèmes. Elle est illustrée par le tableau 11.1. Dans ce tableau $\Phi_{OCaml}$ dénote la fonction de transformation des constructions B vers OCaml.

| Substitutions | Traduction en OCaml |
|---|---|
| PRE *P* THEN *S* END | $\Phi_{OCaml}(S)$ |
| ASSERT *P* THEN *S* END | $\Phi_{OCaml}(S)$ |
| BEGIN *S* END | $\Phi_{OCaml}(S)$ |
| IF *P* THEN *S1*<br>ELSE *S2* END | **if** $\Phi_{OCaml}(P)$<br>**then** $\Phi_{OCaml}(S1)$<br>**else** $\Phi_{OCaml}(S2)$ |
| CASE *E* OF<br>EITHER *T1* THEN *S1*<br>(OR *Ti* THEN *Si*)*<br>[ELSE *S2*] END | **match** $\Phi_{OCaml}$(E) **with**<br>$\Phi_{OCaml}(T1)$->$\Phi_{OCaml}(S1)$<br>(\| $\Phi_{OCaml}(Ti)$->$\Phi_{OCaml}(Si)$)*<br>\|_->$\Phi_{OCaml}(S2)$ |
| VAR *XX* IN $I^3$ ;*S* END | **let** $\Phi_{OCaml}(XX)$=$\Phi_{OCaml}(I)$ **in** $\Phi_{OCaml}(S)$ |
| WHILE *C* DO *S*<br>INVARIANT *P*<br>VARIANT *V* END | **while** $\Phi_{OCaml}$(C) **do** $\Phi_{OCaml}(S)$ |
| *S1 ;S2* | $\Phi_{OCaml}(S1)$ ;$\Phi_{OCaml}(S2)$ |
| *XX := E* | $\Phi_{OCaml}(XX)$ := $\Phi_{OCaml}(E)$ |
| *XX←op(YY)* | proj($\Phi_{OCaml}(XX)$,$\Phi_{OCaml}(op(YY))$) |

TAB. 11.1 – Traduction des structures de contrôle B0 en OCaml

Une opération B se traduira en OCaml par une fonction. Afin de conserver un maximum d'expressivité du code au niveau du profil de la fonction générée, les paramètres de sortie d'une opération B seront traduits en un n-uplet.

#### 11.5.2.2 L'état d'une spécification

Pour les structures de contrôle les informations sont présentes dans l'implantation. Par contre pour déterminer les variables et les constantes, il est nécessaire de prendre les informations dans toute la chaîne de développement du module B à traduire. Les variables et les constantes concrètes peuvent être déclarées et typées dans les abstractions. Les informations nécessaires à la traduction faisant partie des abstractions sont obtenues par dépliage des liens de la clause REFINES.

#### 11.5.2.3 Les types

Les types autorisés pour les données concrètes sont limités par la méthode B à un sous-ensemble des types B existants. Ces types sont appelés types concrets. Ces limitations sont issues des langages cibles initiaux de la méthode B. Il existe deux stratégies par rapport à ce problème :

- se limiter à la liste des types concrets déjà définis [93],
- étendre cette liste par des machines de base.

Selon Petit, la cohabitation de ces deux stratégies ne pose pas de problème.

#### 11.5.2.4 L'utilisation d'assertions

La conservation des propriétés des spécifications jusque dans le code est une nouveauté pour la méthode B. En effet, jusqu'à présent, le code généré est censé être prouvé correct, et donc son exécution être sans erreur. Pour Petit cette vision idéale est trop optimiste. Le code généré est exécuté sur des composants physiques dont les caractéristiques n'ont pas été spécifiées à l'aide d'une méthode formelle. L'ajout d'assertions peut répondre à ce problème en utilisant les propriétés définies dans les invariants de boucles ainsi que les pré-conditions des opérations. Petit utilise la notion de contrat décrite dans [72].

### 11.5.3 De ObjectZ vers C++/Java

Ces deux traductions depuis ObjectZ vers C++ [57] et vers Java [41] utilisent la même stratégie. En effet ObjectZ ne possède pas la notion de substitution. Il est donc nécessaire d'ajouter du code à l'intérieur des structures traduites depuis ObjectZ. La première [57] (de ObjectZ vers C++) utilise une traduction directe. La traduction (fig 11.3) des entités de type produit cartésien ($\mathbb{P}$(Type1 $\times$ Type2)) est réalisée à l'aide de classes templates et d'un système d'héritage. Les classes templates permettent d'utiliser tous les typages spécifiques de la méthode formelle ObjectZ (Type1 et Type2 peuvent eux aussi être des produits cartésiens). Le système d'héritage permet de différencier les fonctions, relations, bijections, surjections, injections et produits cartésiens sans contraintes. Les pré et post-conditions sont utilisées pour remplacer la notion de substitution. Néanmoins il est nécessaire d'ajouter des morceaux de code une fois la traduction effectuée.

```
template <class X, class Y>
class Relation : public Set<Pair<X,Y>>
```

FIG. 11.3 – Traduction d'une relation

La deuxième [41] (de ObjectZ vers Java), ne réalise pas une traduction directe. Il utilise un langage intermédiaire, Jass (Java avec assertion), où il permet au programmeur d'ajouter le code du corps des opérations sachant que les invariants sont traduits dans ce langage et correspondent aux assertions du langage. Le traducteur de Jass vers Java cherche à vérifier le code ajouté en fonction de l'invariant. Dans ce schéma de traduction (ObjectZ vers Java via Jass), la méthode formelle ObjectZ est couplée avec CSP.

### 11.5.4 De VDM++ vers C++ et Java

Les traducteurs de VDM++ vers C++ et Java [52, 54, 53, 55] font partie des VDMTools [4]. La méthode VDM permet de définir un invariant et des pré et post-conditions. Le traducteur VDM++ vers C++ permet de traduire les invariants. Lorsqu'un invariant (fig 11.4) est utilisé pour restreindre la définition d'un type, le type est traduit et un prototype de fonction est issu de la traduction de l'invariant. La portée de la fonction (fig 11.5) est la même que celle du type. Le traducteur ne supportant pas la vérification dynamique de l'invariant, il faut faire manuellement un appel explicite à la fonction de vérification de l'invariant.

$S$ = set of int
inv $s$ == $s$ <> {}

FIG. 11.4 – Un invariant VDM++

```
boolean vdm_A_inv_S(Set);
```

FIG. 11.5 – Traduction de l'invariant VDM++ en une fonction C++

Le traducteur VDM++ vers Java permet de traduire les pré et post-conditions des fonctions. La pré-condition est traduite par une méthode du même nom que la fonction et préfixée par la chaîne de caractères *pre* tandis que la post-condition est préfixée par la chaîne *post*. Le type de retour de ces deux méthodes est **boolean**. Le test de la pré-conditon est exécuté avant l'appel de la fonction et celui de la post-condition est lancé après l'appel de la fonction.

## 11.6 Conclusion

La traduction de B vers C fait apparaître un aplatisseur qui renomme les entités par rapport à leur composant d'origine. Le problème ne se pose pas dans le cadre d'une traduction vers les langages à objets car ceux-ci possèdent la notion de classe équivalente à celle de modularité du langage B. D'autre part la traduction de ObjectZ vers C++ fait apparaître tout un système de classe template non utilisable en Java où il n'existe pas de concept **template** ou équivalent. Cette traduction fait aussi apparaître des structures de données Java relatives aux fonctions, bijections, etc. des spécifications Z. En B ces entités sont raffinées par des fonctions totales utilisées comme des tableaux et traduites sous la même forme dans les

[4] http ://www.ifad.dk

langages impératifs. Le langage VDM++ est très orienté objets, par conséquent l'effort de traduction vers les langages à objets est moindre par rapport à celui qui à pour origine le langage B.

# Chapitre 12

# Génération de code Java à partir de B

## 12.1 Introduction

La génération de code Java définie dans ce chapitre a été réalisée sur la totalité de la syntaxe du B0. Il existe néanmoins une restriction à ce propos dans un cas particulier des ensembles paramètres de machine (cf 12.5.1.1). Nous allons commencer par expliquer les détails de la traduction des composants, du typage spécifique de B et des structures de données. Par la suite nous traiterons la partie opérationnelle : les prototypes et appels d'opérations. Pour finir nous définirons une traduction des substitutions en deux parties qui sont les structures de contrôle et les affectations.

## 12.2 Principes généraux adoptés

### 12.2.1 Le point de départ de la traduction : le B0

Ce chapitre a été élaboré à partir des publications [97, 102]. La traduction d'un projet B vers un langage informatique se fait à partir de machines B qui ont été rendues suffisamment concrètes par le processus de raffinement (fig 12.1) : ces machines sont appelées des implémentations. Elles sont écrites avec un sous ensemble du langage B appelé B0. Le B0 se veut suffisamment proche des langages informatiques pour que la traduction vers ceux-ci en soit facilitée.

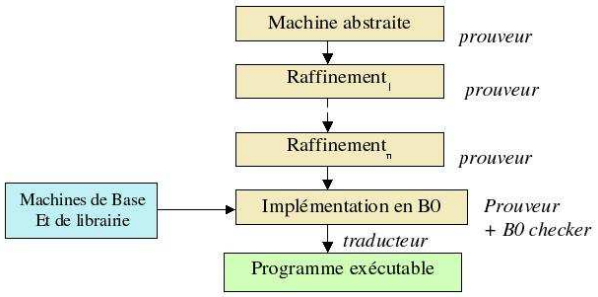


FIG. 12.1 – Cycle de développement d'un projet B

### 12.2.2 Exemple élémentaire de machine abstraite et d'implémentation

Dans une implémentation ou un raffinement, la clause REFINES nomme le composant abstrait qu'elle raffine (fig 12.2).

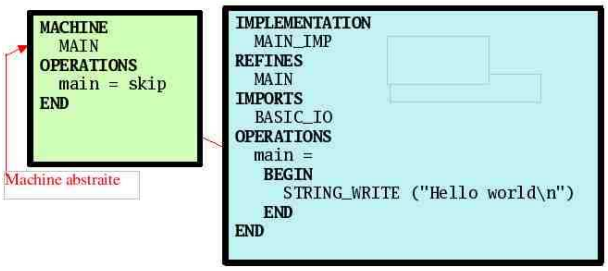


FIG. 12.2 – Machine abstraite et implémentation

### 12.2.3 Schéma de conversion des machines

Un schéma de conversion naturel consiste à traduire chaque implémentation B par une classe. Dans ce cas, les variables concrètes de cette machine sont traduites en variables d'instance de la classe et les opérations en méthodes. Le constructeur par défaut de la classe (ou le mécanisme permettant l'initialisation au moment de l'instanciation) permet de traduire la clause initialisation. Ce schéma de traduction à l'avantage de proposer une gestion simple des instances de machines, la complexité de cette gestion étant déléguée au langage à objet correspondant.

La conversion vers un langage à objet peut bénéficier d'un schéma de conversion dédié aux machines qui ne sont pas instanciées de multiple fois. Il faut que dans ce dernier cas le langage supporte l'utilisation de méthodes et de variables de classes. Dans ce dernier cas, la clause initialisation peut être traduite par une méthode spéciale devant être appelée par la traduction de l'implémentation important cette machine.

#### 12.2.3.1 Stratégie suivie

Dans l'approche que nous suivons, on privilégie une stratégie dans laquelle :
- La traduction d'une machine par une classe permet la génération d'un code source compréhensible et réutilisable par le spécifieur/développeur.
- Les optimisations de bas niveau (remplacement des constantes littérales par les valeurs, ...) sont laissées à la charge du compilateur du langage cible.
- Les optimisations pour savoir s'il faut permettre ou non la création d'instances à partir d'une classe sont gérées par le traducteur de B vers le langage cible.
- Les mécanismes supposés fiables des langages à objets sont utilisés pour disposer d'un support de création d'instances et d'un espace de nommage.

L'utilisation raisonnée des possibilités offertes par les langages à objets permet de ne pas être confronté à des contraintes justifiant la réalisation de traducteurs complexes et donc non fiables.

## 12.3 La représentation d'un projet B

A chaque machine d'implémentation B sera associée une classe. Cette classe peut représenter la machine s'il n'y a qu'une seule instance de celle-ci dans le projet B. Afin de ne pas avoir à gérer les modificateurs d'accès (*public, private, protected* et *package*) pour les fonctions et les variables des classes, un package servant d'espace de noms sera défini et chacune des classes générées appartenant au projet B sera définie dans ce package. Les droits d'accès aux entités seront ceux du package.

## 12.4 La représentation des machines en Java

### 12.4.1 Machines B et schémas de traduction Java

A chaque machine d'implémentation B sera associée une classe. Cette classe peut représenter la machine s'il n'y a qu'une seule instance de celle-ci dans le projet B.

L'utilisation d'une autre machine se fait par la clause `IMPORTS`. De façon générale le lien d'importation permet de créer des instances de machines importées. Le concept d'instance en B est similaire à celui des langages à objets. L'instance d'une machine possède les données et les opérations définies dans cette machine et ses raffinements.

Les particularités de B sont les suivantes :

- Il est possible en B de ne pas donner de nom à une instance, s'il n'y a qu'une seule instance de ce type. Par exemple, si une implantation *M1* instancie une et une seule fois la machine *M2*, et si la machine *M2* possède une opération *f*, alors l'utilisation dans l'implantation *M1* de *f* n'est pas ambiguë.
- Il est nécessaire en B de donner un nom aux instances s'il y a en plus d'une. En effet si *M1* instancie deux fois la machine *M2*, l'utilisation de *f* est ambiguë : à quelle instance de *M2* appartient la fonction *f* ? La solution est de nommer les instances, par exemple *i1* et *i2*, et de préfixer l'appel de la fonction par le nom de l'instance qui contient *f* par exemple : *i1.f*.

Le schéma général est donc d'associer à chaque instance B une instance d'une classe Java. Toutefois dans un but d'optimisation, une machine B qui ne serait instanciée qu'une fois sera représentée par une classe et des variables de classes. Pour réaliser cette optimisation, l'information concernant le nombre d'instances devra être déterminée par le traducteur.

Deux problèmes se posent :

- La machine importée est nommée mais il n'y a qu'une instance : dans la génération de code ce nom disparaît pour ne laisser que le nom de la classe Java correspondante.
- La machine est importée plusieurs fois mais une des instances n'est pas nommée : dans la génération de code il faudra choisir un nom pour cette machine. Le nom choisi fera apparaître le nom de la classe et le fait qu'il s'agit d'une instance anonyme : *anonymous<NomDeClasse>* (fig 12.3 et 12.4).

Chaque classe Java comportera une fonction *INITIALISATION*. Cette fonction sera précédée du modificateur **static** dans le cas d'une classe qui ne sera jamais instanciée. Cette fonction permettra :

- la création des instances de machine en Java via l'opérateur **new** : cet opérateur appelle le constructeur de la classe qui appelle la fonction *INITIALISATION* de l'instance,
- l'initialisation des variables de l'instance ou de la classe.

Dans le cas où la machine est dite statique (les méthodes et variables sont précédées du modificateur **static**), la méthode *INITIALISATION* est appelée non pas par son propre constructeur qui n'existe pas mais par la méthode *INITIALISATION* de la classe qui l'utilise (fig 12.5,12.6). Dans ce cas, on est sûr que la classe dite statique, dont la méthode *INITIALISATION* doit être appelée, sera initialisée même si la classe qui instancie est elle aussi dite statique (il n'y aura pas de constructeur).

### 12.4.2 Exemples

Les exemples qui suivent ont pour but d'illustrer les différents cas de figure qui peuvent se présenter (fig 12.3, 12.4, 12.5 et 12.6).

IMPLEMENTATION
*M1_imp /* implementation de la machine abstraite M1 */*
...
END

IMPLEMENTATION
*M2_imp*
IMPORTS
*i.M1, M1*
...

FIG. 12.3 – Importation nominative ou non d'une machine

## 12.5 Les types

### 12.5.1 Les types de base du langage B

Les types de base sont :
- l'ensemble des entiers relatifs $\mathbb{Z}$,
- l'ensemble des booléens BOOL = {true, false},
- l'ensemble des chaînes de caractères STRING,
- les ensembles énumérés et abstraits, définis dans la clause SETS (voir section suivante).

Pour pouvoir implémenter les données dans un langage informatique, certaines contraintes sont nécessaires :
- Les entiers concrets doivent appartenir à l'intervalle INT. INT est un sous-ensemble de l'ensemble $\mathbb{Z}$, borné par les constantes MININT et MAXINT. La valeur de ces constantes est paramétrable selon la machine cible, de telle sorte que les données soient représentables sur la machine et qu'il n'y ait pas de débordements lors des calculs. Ceci doit se faire au niveau du langage B pour que la preuve en tienne compte.
- Les sous-ensembles d'entiers et les ensembles abstraits doivent être des intervalles d'entiers concrets.

```
class M1 {
    ...
    void INITIALISATION () {
        /* initialisation des variables de M1 */
    }
    M1() {
        INITIALISATION();
    }
}

class M2 { /* On suppose M2 instancié plusieurs fois */
    ...
    M1 i, anonymousM1;
    void INITIALISATION () {
        /* Création des instances */
        i = new M1();
        anonymousM1 = new M1();
        /* initialisation des variables de M2 */
        ...
    }
    M2() {
        INITIALISATION ();
    }
}
```

FIG. 12.4 – Création de deux instances de *M1* l'une n'étant pas nommée

#### 12.5.1.1 Le typage spécifique de B

Les règles de typage de B sont assez éloignées de celles d'un langage de programmation classique. Les différences notoires au niveau des ensembles sont les suivantes :

- l'importation de machines avec des paramètres formels de type ensemble,
- la valuation d'ensembles de machine à l'intérieur de l'implémentation.

Dans les langages de programmation impératifs classiques, une variable est typée et son type ne change plus. Ce n'est pas le cas du langage B. En effet le type d'un constituant peut être réévalué au cours du cycle de raffinement ou de composition des machines (clause IMPORTS). La majorité des difficultés de typage des entités des machines réside dans ces changements de type que l'on appellera recouvrement dans la suite du document.

1. Une machine abstraite peut être paramétrée à l'aide d'un ensemble (fig 12.7). Dans ce cas le type de cette entité est généré à partir du nom de l'ensemble car nous n'avons aucune information de typage sur cette entité. Certaines entités de la machine sont typées à l'aide de cet ensemble. Après une vérification locale à la machine des types, la table des types des entités de la machine $A$ est définie dans le tableau 12.1.

   Ensuite durant l'analyse du projet B, la vérification de types sera effectuée sur la machine $B$ (fig 12.8). Cette machine importe la machine $A$. Lors de l'importation l'ensemble

```
IMPLEMENTATION
    M1_imp /* implementation de la machine abstraite M1 */

OPERATIONS
    ...
    op1≙...
    ...
END

IMPLEMENTATION
    M2_imp /* implementation de la machine abstraite M2 */
...
OPERATIONS
    ...
    op2≙...
    ...
END

IMPLEMENTATION
    M3_imp
IMPORTS
    i.M1, M2
...
OPERATIONS
    ...
    op≙...i.op1 ;...op2 ;...END
    ...
END
```

FIG. 12.5 – Utilisation de machines importées une seule fois

| Entités | Types |
|---|---|
| *ENSEMBLE1* | $\mathbb{P}$*(ENSEMBLE1)* |
| *aa* | *ENSEMBLE1* |
| *dd* | *ENSEMBLE1* |
| *bb* | *ENSEMBLE1* |

TAB. 12.1 – Type des entités de la machine $A$ après vérification locale des types

*ENSEMBLE1* est évalué par l'intervalle *0..255* de type $\mathbb{P}(\mathbb{Z})$. Durant la vérification locale des types sur $B$, le type de *ENSEMBLE1*, $\mathbb{P}$*(ENSEMBLE1)*, doit être remplacé par $\mathbb{P}(\mathbb{Z})$, la table des types des entités de la machine $A$ devient celle du tableau 12.2. Ceci est local et ne pose pas de problème or l'appel de l'opération *op1* de la machine $A$ dans l'opération *op2* de la machine $B$ est réalisée à l'aide du nombre *5*. Cet appel est légal car *5* est de type $\mathbb{P}(\mathbb{Z})$, le même que celui de *ENSEMBLE1* mais la table des types de la

```
class M1 {
    ...
    static void INITIALISATION () { ... }
    /* Bloc exécuté pour l'initialisation de la classe */
    static void op1() { ... }
    ...
}

class M2 {
    ...
    static void INITIALISATION () { ... }
    /* Bloc exécuté pour l'initialisation de la classe */
    static void op2() { ... }
    ...
}

class M3 { /* On suppose M3 instancié une fois */
    void INITIALISATION () {
        /* initialisation des variables de M3 */
        M1.INITIALISATION();
        M2.INITIALISATION();
        ...
    }
    M3() {
        INITIALISATION ();
    }
    ... op () { ... M1.op1(); ...; M2.op2(); ... }
}
```

FIG. 12.6 – Utilisation de deux machines (*M1* et *M2*) non instanciées dans *M3*

machine *A* est pourtant en désaccord avec ce principe car seul le type de *ENSEMBLE1* a été changé.

| Entités | Types |
|---|---|
| *ENSEMBLE1* | $\mathbb{P}(\mathbb{Z})$ |
| *aa* | *ENSEMBLE1* |
| *dd* | *ENSEMBLE1* |
| *bb* | *ENSEMBLE1* |

TAB. 12.2 – Type des entités de la machine *A* après vérification locale partielle des types de *B*

Il est donc nécessaire de modifier le type de toutes les entités de la machine *A* typées à l'aide de *ENSEMBLE1* pour obtenir une table des types à jour comme dans le tableau

```
MACHINE A(ENSEMBLE1)
VARIABLES aa
INVARIANT aa ∈ ENSEMBLE1
OPERATIONS
    bb←op1(dd) ≙
    PRE
        dd ∈ ENSEMBLE1
    THEN
        bb :∈ ENSEMBLE1
    END
END
```

FIG. 12.7 – Exemple de machine avec des paramètres ensemblistes

```
IMPLEMENTATION B
IMPORTS
    A(0..255)
...
OPERATIONS
    cc←op2 ≙
    BEGIN
        cc←op1(5)
    END
END
```

FIG. 12.8 – Exemple d'importation de machines avec des paramètres ensemblistes

12.3.

| Entités | Types |
|---|---|
| *ENSEMBLE1* | $\mathbb{P}(\mathbb{Z})$ |
| *aa* | $\mathbb{Z}$ |
| *dd* | $\mathbb{Z}$ |
| *bb* | $\mathbb{Z}$ |

TAB. 12.3 – Type des entités de la machine $A$ après vérification locale complète des types de $B$

2. Pour la valuation d'ensembles le problème est similaire, on part de la même façon d'une machine $A$ (fig 12.9).
   Dans un raffinement ou une implémentation (fig 12.10) l'entité *ENSEMBLE2* prend le type $\mathbb{P}(\mathbb{Z})$. Le type des variables *aa, bb* et *dd* doit aussi être modifié sinon ceci provoque une erreur car les deux types ne sont pas identiques (le type de *aa, bb* et *dd* passe de *ENSEMBLE2* à $\mathbb{Z}$). L'ensemble n'est valué qu'une seule fois donc le problème est beaucoup moins contraignant (on peut importer une machine plusieurs fois avec des

MACHINE *A*
SETS *ENSEMBLE2*
CONCRETE_VARIABLES
*aa*
INVARIANT
*aa* ∈ *ENSEMBLE2*
...
OPERATIONS
*bb*←*op1(dd)* ≙
PRE
*dd* ∈ *ENSEMBLE2*
THEN
*bb* :∈ *ENSEMBLE2*
END
END

FIG. 12.9 – Exemple de machine avec un ensemble abstrait

MACHINE *A_imp*
REFINES *A*
VALUES
*ENSEMBLE2* = *10..100*
...
OPERATIONS
...
END

FIG. 12.10 – Exemples d'implémentation avec typage d'un ensemble abstrait

valuations d'ensemble différentes mais dans le cycle de raffinement on ne peut valuer un ensemble qu'une seule fois).

Une première solution si ces types sont des classes est d'utiliser la classe `Object` de JAVA dont tous les types héritent mais ces types sont simples (**`int`** et **`boolean`**). Par conséquent il faut utiliser une autre solution qui aurait pu être de créer autant de classes Java que d'instances de machine. Les instances de machines B seraient différenciées par le type utilisé pour valuer l'ensemble. Le problème inhérent à cette stratégie est le renommage de la machine durant la traduction. En effet les machines qui utilisent cette instance ne sauront pas quel identifiant (plusieurs classes Java pour une seule machine B) utiliser pour l'instanciation (si elle est nécessaire) et pour ses méthodes. Une dernière solution a été élaborée. Elle permet de créer une classe avec le type entier (par exemple **`int`**) et de transformer les identifiants booléens {FALSE,TRUE} respectivement en {`0,1`} et bloquer l'utilisation du type STRING pour l'instanciation. De cette manière il n'y aura pas de problème de typage : le seul type utilisé est le type entier.

### 12.5.2 Equivalence entre types B et types Java

Dans un premier temps nous avons essayé de créer des types en B permettant une plus large utilisation des types du langage cible c'est-à-dire Java. Nous avons défini les types *jBint, jBshort* et *jBbyte* comme des intervalles sur INT pour définir les types **`int`**, **`short`** et **`byte`** Java, ce qui permet de définir les équivalences du tableau 12.4.

| TYPE B | TYPE JAVA |
|---|---|
| BOOL | **`boolean`** |
| *jBint* | **`int`** (4 octets, -2147483648..2147483647) |
| *jBshort* | **`short`** (2 octets, -32768..32767) |
| *jBbyte* | **`byte`** (1 octet, -128..127) |
| STRING | **`String`** |

TAB. 12.4 – Proposition d'équivalence non retenue entre les types de base B et les types Java

Trois principaux problèmes nous ont conduit à abandonner cette méthode :
- La mise en application de cette équivalence s'est heurté aux problèmes de recouvrement de types (cf 12.5.1.1) qui oblige à utiliser un type **`int`** ou **`short`** pour tous les types entiers booléens et énumérés.
- Les machines de base telles que *BASIC_IO* utilisent le type B INT. Ce type correspond selon le cas aux types *jBint, jBshort* ou *jBbyte*. Il aurait fallu créer une machine précodée par type cité précédemment. De plus il fallait gérer le choix de l'utilisation d'un type ou d'un autre.
- L'ajout en B des types *jBint, jBbyte* et *jBshort* nous aurait obligé à ajouter les théorèmes permettant la preuve car il n'existe pas de théorèmes connus de l'atelier B facilitant la preuve par rapport à ces types. Les obligations de preuves relatives à l'utilisation de nouveaux types sont de deux sortes :
  - l'utilisation simple de ces types ;
  - les dépassements de capacité.

Nous nous sommes restreints aux types de bases connus en B tels que NAT, INT, BOOL et STRING. Sachant que le but de ce traducteur est de créer des applications pour JavaCard les types utilisés doivent être optimisés. Il est donc nécessaire de pouvoir choisir le type utilisé pour la traduction des types B0. Nous laissons donc le choix à l'utilisateur de traduire les types entiers et booléens par un type **`int`** ou **`short`** (selon l'utilisation définie en 12.5.1.1) en Java selon ses besoins. Ceci est possible par rapport à la preuve en valuant différemment les constantes MININT et MAXINT sur le prouveur comme dans [8].

## 12.6 La Traduction des données

### 12.6.1 Remarques

Il faut faire attention à l'ordre des valuations car il est important. En pratique, il faut traduire les ensembles et les constantes dans l'ordre des valuations (i.e., d'abord les valuations implicites par collage puis celles de la clause VALUES et dans l'ordre de cette clause).

### 12.6.2 Les ensembles

#### 12.6.2.1 Introduction

La clause SETS sert à définir les ensembles énumérés ou abstraits d'un composant. Ces deux types d'ensembles sont composés d'éléments de même type :

1. Un ensemble énuméré, comme son nom l'indique, sert à décrire une énumération de valeurs symboliques.
2. Un ensemble abstrait est utilisé pour désigner des objets dont on ne veut pas définir la structure au niveau de l'abstraction. Tous les ensembles abstraits sont ramenés à un intervalle entier fini non vide, au niveau de l'implantation (clause VALUES). Il y a deux utilisations de ces intervalles :
   - définir de nouveaux types comme dans le paragraphe 12.5.1,
   - définir les bornes inférieures et supérieures d'un tableau.

Dans tous les cas, l'idée est de laisser le compilateur Java réaliser les optimisations. Normalement il devrait pouvoir remplacer les constantes et les expressions constantes par les valeurs numériques correspondantes.

#### 12.6.2.2 Les ensembles abstraits

Les ensembles abstraits se ramènent à des intervalles (fig 12.11) qui sont utilisés pour définir des variables ensemblistes ou des ensembles d'indices de tableau. Ils seront représentés en Java par une classe dans laquelle sera définie en constante la borne inférieure et la borne supérieure. Les intervalles seront représentés par la classe `BRange` qui aura deux champs de type `int` appelés respectivement `borneInferieure` et `borneSuperieure`. Le constructeur de la classe `BRange` permettra d'initialiser ces deux champs (voir fig 12.12).

```
SETS
    VITESSE
VALUES
    VITESSE = 0..100 ;
```

FIG. 12.11 – Un ensemble abstrait en B

Les types BOOL, NATURAL et INT lorsqu'ils seront utilisés en B non pour typer une variable mais comme intervalle seront traduits en Java par une instance de la classe `BRange`. Nous avons choisi le type `int` utilisé dans la traduction de l'ensemble *VITESSE* (fig 12.12) mais le type `short` aurait pu lui aussi être choisi car il dépend du type de base choisi pour la traduction.

#### 12.6.2.3 Les ensembles énumérés

Le langage Java ne possède pas de mécanismes prédéfinis pour écrire une énumération. L'idée consiste ici à définir, comme dans les figures 12.13 et 12.14, une classe par ensemble. Cette classe ne contient que des constantes publiques et ne sera pas instanciée.

```
class BRange {
    final int borneInferieure;
    final int borneSuperieure;
    BRange (int bi, int bs) {
        borneInferieure = bi;
        borneSuperieure = bs;
    }
}

...
    BRange VITESSE = new BRange (0, 100);
...
```

FIG. 12.12 – Traduction d'un ensemble abstrait en Java

SETS
*ETAT* = {*haut, milieu, bas*} ;

FIG. 12.13 – Un ensemble énuméré en B

```
class ETAT {
    static final short haut = 0;
    static final short milieu = 1;
    static final short bas = 2;
    static final short borneInferieure = haut;
    static final short borneSuperieure = bas;
    static Brange getBRange() { ... }
    /* Fonction pour utiliser les ensembles énumérés */
    /* comme arguments de machine */
}
```

FIG. 12.14 – Traduction d'un ensemble énuméré en Java

### 12.6.3 Les constantes concrètes

Les identifiants des constantes concrètes sont énumérés dans une clause CONCRETE_CONSTANTS, sont typés dans une clause PROPERTIES et sont valués dans une clause VALUES (fig 12.15) en B.

Les constantes concrètes correspondent à la définition de constantes, c'est-à-dire une donnée dont la valeur ne varie pas. La notion équivalente existe dans les langages informatiques. La traduction est relativement aisée (fig 12.16) :

- les différents éléments concernant les constantes sont extraits des différentes clauses,
- la constante est définie dans la classe et non dans les instances de machine.

CONCRETE_CONSTANTS
*constante*
PROPERTIES
*constante* $\in$ *NAT*
VALUES
*constante = 10 ;*
*...*

FIG. 12.15 – Une constante en B

```
static final int constante = 10;
```

FIG. 12.16 – Traduction d'une constante en Java

Une constante B peut être un entier, un booléen, un élément d'un ensemble (énuméré ou abstrait), un intervalle fini et non vide ou un tableau .

Remarques :
- La clause CONCRETE_CONSTANTS peut être définie indifféremment dans les machines abstraites, dans les raffinements ou dans les implantations,
- Les valeurs données aux constantes dans les clauses VALUES peuvent faire intervenir des entités (ensembles, constantes) définies dans d'autres machines.

### 12.6.4 Les variables concrètes

Ces variables correspondent à une définition de données modifiables. Elles sont définies en B (fig 12.17) dans la clause CONCRETE_VARIABLES, typées dans la clause INVARIANT et valuées dans la clause INITIALISATION. Une variable concrète B sera traduite en une variable dans un langage informatique. Une variable concrète B peut être un scalaire ou un tableau.

La notion de variable existe dans les langages informatiques. La traduction (fig 12.18) est relativement aisée :
- les différents éléments concernant les variables sont extraits des différentes clauses,
- s'il n'existe qu'une instance de la machine, les variables sont implémentées de façon similaire aux constantes dans la classe à l'exception du mot clé **`final`** : les variables sont implémentées comme des variables de classes,
- s'il existe plusieurs instances de la machine, les variables sont implémentées comme des variables d'instance et le mot clé **`static`** disparaît.

Remarque : comme les constantes concrètes, les variables concrètes peuvent être définies dans les trois types de composants.

### 12.6.5 Les intervalles

La traduction des intervalles est identique à celle des ensembles abstraits : il s'agira d'une instance de la classe *BRange*.

CONCRETE_VARIABLES
*monEntier,*
*monBooleen*
INVARIANT
$monEntier \in INT \wedge$
$monBooleen \in BOOL$
INITIALISATION
*monEntier* := 10 ; *monBooleen* := TRUE

FIG. 12.17 – Des variables en B

```
int monEntier;
int monBooleen;

void INITIALISATION(...) {
    monEntier = 10;
    monBooleen = true;
}
```

FIG. 12.18 – Traduction des variables en Java

### 12.6.6 Les tableaux

#### 12.6.6.1 Introduction

Les tableaux sont des constantes concrètes ou des variables concrètes mais qui présentent suffisamment de particularités pour être traités de façon séparée. L'ensemble des indices d'un tableau peut être un intervalle (défini éventuellement par un ensemble abstrait), un ensemble énuméré ou l'ensemble BOOL.

L'initialisation (la valuation) d'un tableau se fait sur tous les indices par ordre croissant de la borne inférieure à la borne supérieure pour les intervalles, par ordre d'apparition pour les valeurs énumérées, FALSE puis TRUE pour les valeurs booléennes.

Remarque : les tableaux peuvent servir à implémenter des ensembles.

#### 12.6.6.2 Tableaux constants à une dimension avec des intervalles

Les tableaux constants à une dimension sont définis en B (fig 12.19) comme les constantes à l'aide des clauses CONCRETE_CONSTANTS, PROPERTIES et VALUES. La traduction ne fait pas apparaître des tableaux constants dans le cas où il est initialisé avec un produit cartésien car l'espace qu'il utilise doit lui être alloué avant de l'affecter par des valeurs (fig 12.20).

CONCRETE_CONSTANTS
*tableau1, tableau2, tableau3, tableau4*
PROPERTIES
*tableau1* $\in$ *(0..2)* $\rightarrow$ INT $\wedge$ */* Un tableau commençant à l'indice 0 */*
*tableau2* $\in$ *(-1..1)* $\rightarrow$ NAT $\wedge$ */* Un tableau commençant à l'indice -1 */*
*tableau3* $\in$ *(0..6)* $\rightarrow$ INT $\wedge$ */* Tableau initialisé avec une valeur */*
*tableau4* $\in$ *(2..8)* $\rightarrow$ INT
VALUES
*tableau1* = $\{0 \mapsto 2, 1 \mapsto 4, 2 \mapsto -6\}$ ;
*tableau2* = $\{-1 \mapsto 3, 0 \mapsto 5, 1 \mapsto 7\}$ ;
*tableau3* = *(0..6)*{6}* ;
*tableau4* = *(2..8)*{9}* ;

FIG. 12.19 – Constantes tableaux à une dimension en B

```
static final int[] tableau1 = {2, 4, -6};
static final int[] tableau2 = {3, 5, 7};
static final int[] tableau3 = new int [6-(0)+1];
static final int[] tableau4 = new int [8-(2)+1];

void INITIALISATION() {
    Arrays.fill(tableau3, 6);
    Arrays.fill(tableau4, 9);
}
```

FIG. 12.20 – Traduction de constantes de type tableaux en Java

CONCRETE_VARIABLES
*tableau1, tableau2, tableau3, tableau4*
INVARIANT
*tableau1* $\in$ *(0..2)* $\rightarrow$ INT $\wedge$ */* Un tableau commençant en 0 */*
*tableau2* $\in$ *(-1..1)* $\rightarrow$ NAT $\wedge$ */* Un tableau commençant en -1 */*
*tableau3* $\in$ *(0..6)* $\rightarrow$ INT $\wedge$ */* Tableau initialisé avec une valeur */*
*tableau4* $\in$ *(2..8)* $\rightarrow$ NAT
INITIALISATION
*tableau1* := $\{0 \mapsto 2, 1 \mapsto 4, 2 \mapsto -6\}$ ;
*tableau2* := $\{-1 \mapsto 3, 0 \mapsto 5, 1 \mapsto 7\}$ ;
*tableau3* := *(0..6)*{6}* ;
*tableau4* := *(2..8)*{9}* ;

FIG. 12.21 – Variables de type tableaux en B

```
int[] tableau1 = {2, 4, -6};
int[] tableau2 = {3, 5, 7};
int[] tableau3 = new int[6-(0)+1];
int[] tableau4 = new int[8-(2)+1];

void INITIALISATION() {
    Arrays.fill(tableau3, 6 );
    Arrays.fill(tableau4, 9);
}
```

FIG. 12.22 – Traduction des variables de type tableaux en Java

#### 12.6.6.3 Tableaux variables à une dimension avec des intervalles

Le mot clé static serait utilisé s'il n'y avait qu'une instance de la machine. Un élément de tableau est accessible dans les opérations B par une syntaxe de la forme *identifiant(index)* (fig 12.21). Un tel élément de tableau sera traduit en java par `identifiant[index - borneInferieure]`. Par exemple `tableau4(3)` vaut *9* dans la figure 12.22 et sera traduit en Java par `tableau4[3-2]`.

#### 12.6.6.4 Tableaux dont les indices sont des valeurs énumérées ou booléennes

La traduction des Tableaux dont les indices sont des valeurs énumérées ou booléennes (fig 12.23) sont traduits d'une manière particulière. Il est nécessaire d'utiliser des entiers comme indices, par conséquent nous traduisons le type énuméré ou booléen par des ensembles d'entiers (fig 12.24).

SETS
  *Enumere* = {*e1*, *e2*, *e3*, *e4*}
CONCRETE_CONSTANTS
  *tableau1*, *tableau2*
PROPERTIES
  $tableau1 \in Enumere \rightarrow \text{INT} \wedge$
  $tableau2 \in \text{BOOL} \rightarrow \text{INT}$
VALUES
  $tableau1 = \{e1 \mapsto 0, e2 \mapsto 1, e3 \mapsto 2, e4 \mapsto 3\}$ ;
  $tableau2 = \{\text{FALSE} \mapsto 0, \text{TRUE} \mapsto 1\}$ ;

FIG. 12.23 – Tableaux dont les indices sont des énumérés ou des booléens en B

#### 12.6.6.5 Tableaux pour représenter des variables ensemblistes

Les tableaux peuvent servir à implémenter des ensembles comme dans l'exemple de la figure 12.25.

```
static final int[] tableau1 = {0, 1, 2, 3};
static final int[] tableau2 = {0, 1};
```

FIG. 12.24 – Traduction des tableaux dont les indices sont des énumérés ou des booléens B en Java

MACHINE
*M1*
...
SETS
*VITESSE*
ABSTRACT_VARIABLES
*Vitesse* $\subset$ *VITESSE*
...
END

IMPLEMENTATION
...
VALUES
*VITESSE = 0..100 ;*
CONCRETE_VARIABLES
*VitesseImp*
INVARIANT
*VitesseImp* $\in$ *VITESSE* $\rightarrow$ BOOL
...
END

FIG. 12.25 – Tableau qui représente une variable ensembliste

#### 12.6.6.6 Tableaux de tableaux

Les tableaux de tableaux (fig 12.26) sont traduits en Java par des tableaux à plusieurs dimensions (fig 12.27).

### 12.6.7 Les paramètres des machines

#### 12.6.7.1 Les paramètres formels

Les paramètres formels (fig 12.28) des machines B peuvent être :
- des ensembles abstraits,
- des éléments typés dans la clause CONSTRAINTS,
- des instances de booléens, d'entiers,
- des éléments appartenant à un ensemble abstrait.

CONCRETE_CONSTANTS
*tableau1, tableau2*
PROPERTIES
$tableau1 \in (0..2) \rightarrow ((0..1) \rightarrow \text{INT}) \wedge$
$tableau2 \in (0..1)*(0..3) \rightarrow \text{INT}$
VALUES
*tableau1* = {
$0 \mapsto \{0 \mapsto -12, 1 \mapsto 0\}$,
$1 \mapsto \{0 \mapsto -13, 1 \mapsto 1\}$,
$2 \mapsto \{0 \mapsto -14, 1 \mapsto 2\}$
};
*tableau2* = (0..1)*(0..3)*5 ;

FIG. 12.26 – Tableaux de tableaux en B

```
static final int[][] tableau1 = {{-12,0},{-13,1},{-14,2}};
static final int[][] tableau2 = new int[1-(0)+1][3-(0)+1];
static {
    for (int i = 0; i <= 1; i += 1) {
        Arrays.fill(tableau2[i], 5);
    }
}
```

FIG. 12.27 – Traduction en Java de tableaux de tableaux

#### 12.6.7.2 Les paramètres effectifs

Les paramètres effectifs (fig 12.28) qui suivent le nom de la machine dans la clause IMPORTS peuvent être :
- Des scalaires : des entiers, des booléens, des valeurs d'un ensemble énuméré
- Des ensembles : énumérés, intervalles sur les entiers, prédéfinis (BOOL, NAT et INT)

#### 12.6.7.3 La traduction en JAVA

Ces différents paramètres apparaissent dans l'ordre dans la fonction `INITIALISATION` (fig 12.29). Cette fonction est appelée soit par le constructeur de la classe qui reçoit lui-même les paramètres et se contente de les répercuter dans le cas du multi-instance, soit directement dans le cas d'une instance unique.

## 12.7 Les opérations

### 12.7.1 La traduction des opérations B et des paramètres d'entrée

La clause OPERATIONS (fig 12.30) permet de définir les opérations d'un composant. Les opérations définissent la partie dynamique du langage B. Elles peuvent faire évoluer les

```
IMPLEMENTATION
    M1_imp (capacite, MONENSEMBLE)
    /* implementation de la machine abstraite M1 */
...
END

IMPLEMENTATION
    M2_imp
IMPORTS
    i.M1 (10, 2..8), M1(30, 6..9)
...
```

FIG. 12.28 – Création de plusieurs instances paramétrées en B

```
class M1 {
...
    void INITIALISATION (int capacite, BRange MONENSEMBLE) {
        ...
    }
    M1(int capacite, BRange MONENSEMBLE) {
        /* initialisation des variables de M1 */
        ...
        INITIALISATION(capacite, MONENSEMBLE);
    }
}

class M2 {
    ...
    M1 i, anonymousM1;
    void INITIALISATION () {
        /* initialisation des variables de M2 */
        ...
        /* Création des instances */
        i = new M1(10, new BRange(2,8));
        anonymousM1 = new M1(30, new BRange(6,9));
    }
    M2() {
        INITIALISATION ();
    }
}
```

FIG. 12.29 – Traduction de plusieurs instances paramétrées en Java

variables du composant. Une opération peut avoir des paramètres d'entrée et de sortie (les résultats).

Une opération est composée de deux parties :
- l'en-tête définit le nom de la fonction ainsi que les paramètres formels d'entrée et/ou de sortie éventuels,
- le corps de l'opération est une substitution (Section 3.6) qui correspond à l'implantation de cette opération.

Nous nous intéressons ici uniquement à la traduction de l'en-tête de l'opération. Une opération B se traduit naturellement par une fonction Java. Pour les paramètres d'entrée, les différents cas sont les suivants :
- le type du paramètre est un type de base B alors celui-ci est traduit en une variable Java possédant le même nom et dont le type est la traduction du type de base B.
- Si le type d'un paramètre B est un intervalle, le type choisi en Java est le type de base englobant.
- Si le paramètre est une valeur d'un ensemble énuméré, le type choisi en Java est **`int`** ou **`short`** selon le type de base choisi par l'utilisateur.

### 12.7.2 Le problème des résultats multiples

La traduction d'une opération en Java pose une difficulté majeure, dans le cas où les résultats sont au nombre de deux ou plus. La difficulté réside dans le fait que le langage Java ne gère pas les retours multiples. Une fonction Java peut soit retourner **`void`** (pas de retour) soit un type de base ou un objet. Encore une fois, nous avons le choix entre plusieurs solutions :
- Mettre les paramètres de sortie dans les paramètres d'entrée. Pour que cette solution marche, il faut que les paramètres soient des objets car Java transmet les arguments par valeur et les valeurs de sortie ne seraient donc jamais mises à jour.
- Retourner un tableau qui contiendrait les différentes valeurs de retour. Un autre problème se pose : pour contenir indifféremment n'importe quel type, un tel tableau ne peut contenir que des objets de type *`Object`*. Ceci implique une conversion des types vers leur classe enveloppe. A l'inverse, une conversion de la classe enveloppe vers le type de base correspondant est nécessaire après le retour.
- Restreindre en B le nombre de résultats possibles à 1 au maximum.
- Travailler avec des variables globales.

La solution choisie est la suivante (fig 12.31) :
- Le premier résultat B est renvoyé comme le résultat normal de la fonction Java
- Les autres résultats B deviennent des paramètres d'entrée de la fonction Java : les types scalaires étant encapsulés dans leur type enveloppe (INT dans *`BInteger`*, NAT dans *`BInteger`* et BOOL dans *`BBoolean`*).

L'avantage de cette solution est de permettre d'être efficace dans le cas d'environnements contraints comme une carte à puce (le spécifieur B doit s'abstenir dans ce cadre d'utiliser plus d'un résultat par opération) et de rester proche d'une programmation classique en Java. Dans le cas où l'environnement cible possède des ressources suffisantes, les possibilités offertes permettent d'utiliser l'ensemble du langage B.

### 12.7.3 Un exemple

Dans l'exemple de la figure 12.30, l'opération B *foo* retourne trois résultats mais ne prend pas de paramètres. La traduction de cette opération (fig 12.31) fait apparaître un seul résultat, ceci est imposé par les concepts Java, et deux paramètres qui résultent de la traduction des résultats numéro deux et trois de l'opération B.

IMPLEMENTATION
*M_imp*
CONCRETE_VARIABLES
*var1,*
*var2*
INVARIANT
*var1* ∈ INT ∧
*var2* ∈ INT
INITIALISATION
*var1* := *10* ;
*var2* := *20*
OPERATIONS
*bb*←*test(v)* ≙
BEGIN *bb* := BOOL(*var1* = *v* ) END ;
*augmenter* ≙
BEGIN *var2* := *var2* + *1* END ;
*v1,v2,bb*←*foo* ≙
BEGIN *v1* := *var1* ; *v2* := *var2* ; *bb* := BOOL(*var1* = *var2*) END
END

FIG. 12.30 – Un exemple de passage de paramètres en B

### 12.7.4 Retour sur la sémantique du passage de paramètres en B

Selon [28] en B, les opérations sont déclarées au niveau des machines par un texte de la forme :

$r \leftarrow op(p) \mathrel{\widehat{=}}$ PRE $P$ THEN $S$ END

où $p$ est une liste de paramètres formels d'entrée et $r$ est une liste de paramètres formels de sortie. Le prédicat $P$ est la précondition et $S$ la substitution corps de l'opération. Un appel de l'opération $op$ s'écrit : $v \leftarrow op(e)$ où $v$ est la liste des paramètres effectifs de sortie (une liste de variables) et $e$ la liste des paramètres effectifs d'entrée (une liste d'expressions). La sémantique de l'appel d'opération dans le B-Book [3] est définie par la règle de substitution suivante :

(1) : $v \leftarrow op(e) \equiv$ [r,p := v,e] PRE $P$ THEN $S$ END

```
class M {
    ...
    boolean test(int v) {
        return (var1 == v);
    }

    void augmenter() {
        var2 := var2 + 1;
    }

    int foo(BInteger v2, BBoolean bb) {
        v1 = var1;
        v2.setValue (var2);
        bb.setValue (var1==var2);
        return (v1);
    }
}
```

FIG. 12.31 – Traduction du passage de paramètres en Java

Les restrictions sur l'appel sont que $v$ est une liste de variables simples (et non des références à des éléments de tableau, par exemple) et que ces variables n'apparaissent pas dans $P$ et $S$ (non aliasing). La définition (1) est adaptée à la preuve, mais n'est pas opérationnelle pour la traduction. En effet, un appel d'opération en B doit pouvoir se traduire par un appel au code de cette opération. La règle ci-dessus n'est directement applicable que sur les substitutions du niveau machine, qui spécifie un changement d'état de manière atomique. Dans le cas des implémentations, qui peuvent contenir des séquencements ou des itérations, il faut utiliser la règle suivante :

(2) : $v \leftarrow op(e) \equiv$ PRE [p :=e] $P$ THEN VAR $p,r$ IN $p := e ; S ; v := r$ END END

qui correspond explicitement au passage par copie. Les restrictions sur la structuration des espaces de variables en B font que la règle (2) est équivalente à la forme :

(3) : $v \leftarrow op(e) \equiv$ PRE $[p := e]P$ THEN VAR $p$ IN $p := e ; [r := v]S$ END END

La raison essentielle de l'équivalence entre (2) et (3) est que les variables de la liste $v$ n'apparaissent pas dans $S$. il s'ensuit que les paramètres résultats peuvent être substitués, ce qui correspond à un passage par référence. C'est cette règle qui est utilisée par les traducteurs de l'Atelier B. Un dernier résultat est que les paramètres d'entrée peuvent aussi être passés par référence sous la condition :

- la liste des paramètres effectifs d'entrée ne contient aucune occurrence des paramètres effectifs résultats, ni aucune occurrence des variables référencées dans l'opération.

Sous ces hypothèses, il peut être montré que la règle (1) est applicable sur les substitutions admises dans les implémentations. La preuve est détaillée dans [80]. Cette optimisation permet

de traduire les appels d'opérations B par des passages par référence, ce qui est intéressant dans le cas de paramètres de type tableau par exemple, car la copie est évitée.

Autrement dit les appels d'opération de la figure 12.32 sont interdits. Avant la création

$x \leftarrow op(x)$
$x, x \leftarrow op(...)$
$op\ (x, x)$

FIG. 12.32 – Appels d'opération interdits

des outils réalisés dans le cadre du projet BOM, le passage des tableaux (fig 12.33) se faisait par copie et il était impossible de modifier un tableau passé comme paramètre d'entrée. Pour être plus réaliste vis à vis des applications en terme de développement et d'économie des ressources les paramètres tableaux seront passés par référence (fig 12.34) en respectant les règles indiquées juste avant.

```
MACHINE
...
OPERATIONS
    T6, T5 ← op(T4) ≙
    PRE T4 : 1..4 → BOOL
    THEN
        /* T4 n'est pas accessible en écriture
        T4(2) := TRUE ||
        */
        T5 := T4 ||
        T6 := T4
    END
END

IMPLEMENTATION
...
OPERATIONS
    T6, T5 ← op(T4) ≙
    BEGIN
        /* T4 n'est pas accessible en écriture
        T4(2) := TRUE ||
        */
        T5 := T4 ;
        T6 := T4
    END
END
```

FIG. 12.33 – Un exemple de passage d'un paramètre tableau par référence

```
class ... {
    ...
    boolean[] op( boolean[] T4, boolean[] T5) {
        boolean[] T6 = new boolean[T4.length];
        /* T4 n'est pas accessible écriture
        T4[2-(1)] = true;
        */
        System.arraycopy (T4, 0, T5, 0 );
        System.arraycopy (T4, 0, T6, 0);
        return (T6);
    }
}
```

FIG. 12.34 – Traduction en Java d'un paramètre tableau passé par référence

## 12.8 Les substitutions

### 12.8.1 Introduction

Une substitution est une notation mathématique permettant de modéliser la transformation d'états par une transformation de prédicats. Dans les opérations, le corps est une substitution. Au niveau du B0 les substitutions sont appelées instructions dans la terminologie de ClearSy pour montrer le lien avec un langage de programmation. Les substitutions présentées ci-dessous correspondent aux substitutions à partir desquelles du code source peut être généré.

<Instruction> : :=
<InstructionDevientEgal>
| <InstructionAppelOperation>

| <InstructionSequence>
| <InstructionBloc>
| <InstructionIdentite>
| <InstructionVariableLocale>

| <InstructionConditionnelle>
| <InstructionCas>
| <InstructionTantQue>
| <InstructionAssertion>

Le tableau 12.5 représente la traduction des structures de contrôle de B0 en Java. Dans ce tableau S, S1, S2, S3 et Si sont des substitutions, T1 et Ti des valeurs, C, P, P1, P2 et V des prédicats, E une expression, XX un identifiant de variable et I une instruction qui permet en B0 de typer les variables locales. *Type* et $\Phi_{Java}$ dénotent respectivement la fonction d'extraction du type d'une instruction et la fonction de transformation des constructions B vers Java.

| Substitutions | Traduction en Java |
|---|---|
| PRE *P* THEN *S* END | $\Phi_{Java}(S)$ |
| ASSERT *P* THEN *S* END | $\Phi_{Java}(S)$ |
| BEGIN *S* END | $\Phi_{Java}(S)$ |
| IF *P1* THEN *S1*<br>(ELSIF *P2* THEN *S2*)*<br>[ELSE *S3* END] | **if** $(\Phi_{Java}(P1))\{\Phi_{Java}(S1)\}$<br>**(else if** $(\Phi_{Java}(P2))$ $\{\Phi_{Java}(S2)\}$**)***<br>[**else** $\{\Phi_{Java}(S3)\}$] |
| CASE *E* OF<br>EITHER *T1* THEN *S1*<br>(OR *Ti* THEN *Si*)*<br>[ ELSE *S2*] END | **switch** $(\Phi_{Java}(E))$ {<br>**case** $\Phi_{Java}(T1)$ : $\Phi_{Java}(S1)$ **break;**<br>**(** **case** $\Phi_{Java}(Ti)$ : $\Phi_{Java}(Si)$ **break;** **)***<br>[ **default :** $\Phi_{Java}($ *S2*) **break;**] } |
| VAR *XX* IN *I;S* END | $\{\Phi_{Java}(\text{Type}(I))$ $\Phi_{Java}(XX)$ ;$\Phi_{Java}(\text{I})$ **;** $\Phi_{Java}(S)\}$ |
| WHILE *C* DO *S*<br>INVARIANT *P*<br>VARIANT *V* END | **while (**$\Phi_{Java}(C)$**)** $\{\Phi_{Java}(S)\}$ |
| *S1 ;S2* | $\Phi_{Java}(S1)$ ;$\Phi_{Java}(S2)$ |
| *XX := E* | $\Phi_{Java}(XX)$ := $\Phi_{Java}(E)$ |

TAB. 12.5 – Traduction des structures de contrôle B0 en Java

### 12.8.2 La substitution *Devient égal*

La substitution := (*Devient égal*) remplace une variable par une expression. Elle correspond donc à une affectation. Les différentes possibilités sont les suivantes :

- affectation d'une donnée entière,
- affectation d'une donnée booléenne à l'aide d'une condition,
- affectation d'un élément de tableau,
- affectation d'une donnée tableau.

<InstructionDevientEgal> : :=
    <Variable> := <ExpressionScalaire>
    | <Tableau> := <ExpressionTableau>

Dans le cas simple, la traduction en Java revient à remplacer le := par =. Dans le cas des tableaux à une dimension on utilisera la fonction de copie (arraycopy) de la classe System (fig 12.35 et 12.36). Dans le cas des tableaux à deux dimensions, une itération sera effectuée sur la première dimension et la fonction de copie sera utilisée pour la deuxième dimension.

### 12.8.3 Substitution appel d'opération

Elle permet d'appliquer la substitution d'une opération en remplaçant les paramètres formels par des paramètres effectifs (fig 12.37). Les paramètres d'entrée, s'il y en a, sont des expressions scalaires, des tableaux passés par référence et des chaînes littérales. Les paramètres de sortie éventuels sont des variables scalaires ou un tableau.

<InstructionAppelOperation> : :=
    [((<Variable> | <Tableau>) (',' (<Variable> | <Tableau> ))*) '←']

MACHINE
*Tableaux*
CONCRETE_VARIABLES
*T1, T2, T3*
INVARIANT
$T1 \in -5..5 \rightarrow \text{INT} \wedge$
$T2 \in -5..5 \rightarrow \text{INT} \wedge$
$T3 \in (1..2)*(1..2) \rightarrow \text{BOOL}$
INITIALISATION
$T1 :\in -5..5 \rightarrow \text{INT} \parallel$
$T2 :\in -5..5 \rightarrow \text{INT} \parallel$
$T3 :\in (1..2)*(1..2) \rightarrow \text{BOOL}$
...

IMPLEMENTATION
*Tableaux_imp*
REFINES
*Tableaux*
INITIALISATION
*T1 := (-5..5)*1 ;*
*T2 := T1 ;*
$T3 := \{1 \mapsto 1 \mapsto \text{TRUE}, 1 \mapsto 2 \mapsto \text{FALSE}, 2 \mapsto 1 \mapsto \text{FALSE}, 2 \mapsto 2 \mapsto \text{TRUE}\}$
...
OPERATIONS
...
*var := T3(1↦2) ;*
*T1(2) := T2(5) ;*
...
END

FIG. 12.35 – Substitution *Devient égal* pour les tableaux

<OperationName>
['('<ExpressionScalaire> | <Tableau> | <String>)
(','(<ExpressionScalaire> | <Tableau> | <String>))* ')' ]

En Java, cette substitution correspond à un appel de méthode. Le ← lorsqu'il existe est traduit par =. Etudions sur quelques exemples les différentes possibilités (fig 12.37 et 12.38).

L'appel de fonction subit les mêmes contraintes que la définition de méthodes (clause OPERATIONS), en ce qui concerne les retours multiples. Par conséquent, il est nécessaire de faire une conversion des classes enveloppes vers les types primitifs de Java. Au niveau des résultats encapsulés, il y a un numéro postfixé pour éviter les redéclarations si une variable est utilisée plusieurs fois comme résultat d'opération.

```
class Tableaux {

    int[] T1 = new int[5-(-5)+1];
    int[] T2 = new int[5-(-5)+1];
    boolean[][] T3 = {{ true, false}, {false, true}};

    void INITIALISATION() {
        Arrays.fill (T1, 1);
        System.arraycopy (T1, 0, T2, 0);
...
    }

    ...
    var = T3[1-(1)][2-(1)];
    T1[2-(-5)] = T2[5-(-5)];
    ...
}
```

FIG. 12.36 – Traduction en Java de la substitution *Devient égal* pour les tableaux

### 12.8.4 Substitution *Séquencement*

La substitution *Séquencement* correspond à l'application en séquence de deux substitutions. Traduire le séquencement est simple : il suffit de traduire les deux substitutions qu'il contient, dans l'ordre d'apparition (fig 12.39 et 12.40).

<InstructionSequence> : := <Instruction> ';' <Instruction>

### 12.8.5 Substitution *Bloc*

Elle sert à grouper plusieurs substitutions qui seront effectuées en séquence. Elle sert de parenthèse, en quelque sorte. Elle est nécessaire dans le cas de plusieurs substitutions et facultative dans le cas d'une substitution unique.

<InstructionBloc> : := BEGIN <Instruction> **(** ';' <Instruction> **)*** END
En Java, le principe est le même : les blocs, représentés par un couple d'accolades ({}), ne sont obligatoires que pour un bloc d'au moins deux instructions. Cependant, dans un souci de clarté, il est préférable de mettre systématiquement les accolades.

### 12.8.6 Substitution *Identité*

Cette substitution ne fait rien. Elle est utilisée par exemple pour indiquer que certaines branches d'une instruction IF ou CASE ne modifient pas les variables. Elle est traduite si nécessaire en java par un bloc vide.

<InstructionBloc> : := *Skip*

IMPLEMENTATION
*M_imp*
CONCRETE_VARIABLES
*res1,res2, res3*
IMPORTS
*i.M2*
INVARIANT
$res1 \in INT \wedge$
$res2 \in 0..4 \rightarrow INT \wedge$
$res3 \in BOOL$
INITIALISATION
*res1 = 10 ;*
*res2 = (0..4)*{-1} ;*
*res3* = TRUE
OPERATIONS
*foo* $\widehat{=}$
BEGIN
*res1*$\leftarrow$*i.op1(res2) ;*
*i.op2 (1,* TRUE*) ;*
*res2,res3*$\leftarrow$*i.op3*
END
...

FIG. 12.37 – La substitution appel d'opération en B

### 12.8.7 Substitution *Variable locale*

Dans les opérations, il est parfois nécessaire d'utiliser des variables qui ne sont pas définies au niveau de la clause CONCRETE_VARIABLES. La substitution variable locale permet, comme son nom l'indique, de définir une liste des variables locales à l'opération (fig 12.41). Ces variables ne sont pas typées lors de leur déclaration mais lors de leur première affectation.

<InstructionBloc> : :=
VAR <BIdentifier> (',' <BIdentifier>)*
IN <Instruction> END

La traduction en Java se fait par un bloc et les variables locales sont déclarées au début de ce bloc (fig 12.42).

### 12.8.8 Substitution *Condition*

La substitution conditionnelle IF définit un ou plusieurs comportements en fonction de la validité d'une ou plusieurs conditions. Sa syntaxe est la suivante :

<InstructionConditionnelle> : :=
IF <Condition> THEN <Instruction>

```
class M {

    int res1;
    int[] res2 = new int[4-(0)+1];
    M2 i;
    ...
    void foo() {

        res1 = i.op1(res2);
        i.op2(1, true);
        BBoolean res3__Object = new BBoolean();
        res2 = i.op3(res3__Object);
        res3 = res3__Object.getValue();
    }
    ...
}
```

FIG. 12.38 – Traduction de la substitution appel d'opération en Java

```
tmp := x;
x := y;
y := tmp;
```

FIG. 12.39 – Le séquencement B

```
tmp = x;
x = y;
y = tmp;
```

FIG. 12.40 – Traduction en Java du *Séquencement* B

```
VAR bb, ii IN
    bb := unBooleen;
    ii := 2 + unEntier;
    ...
END
```

FIG. 12.41 – Les substitutions *Variables locales* en B

(ELSIF <Condition> THEN <Instruction>)*
[ ELSE <Instruction> ]

```
{
    boolean bb;
    int ii;

    bb = unBooleen;
    ii = 2 + unEntier;
    ...
}
```

FIG. 12.42 – Traduction des substitutions *Variables locales* B en Java

END

Elle peut prendre plusieurs formes :
- IF $p$ THEN $s$ END : la forme minimale. La substitution $s$ est appliquée si $p$ est vérifié.
- IF $p$ THEN $s1$ ELSE $s2$ END : la substitution $s1$ est appliquée si $p$ est vérifié. Dans le cas contraire, c'est la substitution $s2$ qui est appliquée.
- IF $p1$ THEN $s1$ ELSIF $p2$ *THEN* $s2$ ... ELSIF $pn$ THEN $sn$ END : cette forme est équivalente à imbriquer plusieurs substitutions IF. La substitution $s1$ est appliquée si $p1$ est vérifié, $s2$ si $p2$ est vérifié, etc...
- IF $p1$ [THEN $s1$ ELSIF $p2$ THEN $s2$ ... ELSIF $pn$ THEN $sn$ ELSE $sd$ END : même définition que précédemment, à l'exception qu'il existe une substitution par défaut qui est appliquée au cas où aucun prédicat n'est vérifié.

Tous les langages de programmation possèdent une structure de choix équivalente que l'on peut voir sur les deux exemples suivants du tableau 12.6.

| | |
|---|---|
| IF $a < b$ THEN<br>  $a := b$<br>END | `if (a < b) {`<br>`  a = b;`<br>`}` |
| IF $a < b$ THEN<br>  $b := 0$<br>ELSIF $a > b$ THEN<br>  $a := 0$<br>ELSE<br>  $a := b/2$<br>END | `if (a < b) {`<br>`  b = 0`<br>`} else if (a > b) {`<br>`  a = 0;`<br>`} else {`<br>`  a = b/2;`<br>`}` |

TAB. 12.6 – Substitution *Condition* en B (à gauche) et en Java (à droite)

On peut remarquer que des accolades délimitent les blocs même si ce n'est pas nécessaire. A noter aussi les parenthèses obligatoires autour des prédicats.

### 12.8.9 Substitution *Condition par cas*

Cette deuxième substitution conditionnelle permet de choisir entre divers comportements en fonction de la valeur d'une expression (fig 12.43). Tout comme la substitution IF, la structure de choix existe dans la plupart des langages de programmation : en Java, c'est l'instruction `switch` (fig 12.44).

<InstructionConditionnelle> : :=
CASE <ExpressionScalaire> OF
EITHER <ConstanteScalaire> THEN <Instruction>
(OR <ConstanteScalaire> (',' <ConstanteScalaire>)* THEN <Instruction>)*
[ELSE <Instruction>]
END
END

CASE *x/5+3* OF
EITHER *3* THEN
*y := 0*
OR *4,5,6* THEN
*y := x*
OR *10* THEN
*y := 10*
ELSE
*y := -x*
END
END

FIG. 12.43 – La substitution *Condition par cas* en B

### 12.8.10 Substitution *Boucle tant que*

Cette substitution réalise une itération sur un bloc. Les instructions contenues dans ce bloc sont effectuées tant qu'une condition (un prédicat) est vérifiée. Lorsque la condition n'est plus vraie, l'exécution se poursuit avec l'instruction suivant la boucle (fig 12.45).

<InstructionTantQue> : :=
WHILE <Condition> DO
INVARIANT <Predicat>
VARIANT <Expression>
END

Les langages informatiques possèdent tous une structure équivalente.

Dans l'exemple ci-dessus (fig 12.46), nous voyons que les parties VARIANT et INVARIANT de la substitution n'influent pas sur la traduction. En fait, ces deux clauses servent à la preuve, pour vérifier notamment que la boucle se termine forcément.

```
switch (x/5+3) {
    case 3 :
        y = 0;
        break;
    case 4 :
    case 5 :
    case 6 :
        y = x;
        break;
    case 10 :
        y = 10;
        break;
    default :
        y = -x;
}
```

FIG. 12.44 – Traduction en Java de la substitution *Condition par cas* de B

*i* := 0;
WHILE *i* < 5 DO
    *var* := *var* * 2;
    *i* := *i* + 1
INVARIANT
    *i* ∈ NAT ∧
    *var* ∈ NAT
VARIANT
    5 - *i*
END

FIG. 12.45 – La substitution *Boucle tant que* en B

```
i = 0;
while (i < 5) {
    var = var * 2;
    i = i + 1;
}
```

FIG. 12.46 – Traduction de la substitution *Boucle tant que* en Java

### 12.8.11 Substitution *Assertion*

Dans cette instruction dont la syntaxe est donnée ci-dessous, seule la partie <Instruction> est traduite par un bloc Java car on fait l'hypothèse que les composants B auront été

préalablement prouvés.

<InstructionAssertion> : :=
ASSERT <PREDICAT> THEN <Instruction> END

Remarque : on est limité par les caractéristiques du langage cible malgré les dispositifs trouvés pour pallier le manque de flexibilité du langage Java par rapport à B.

# Chapitre 13

# Vers une traduction pour JavaCard

## 13.1 Introduction

Ce chapitre vise à expliquer les principales différences entre une traduction Java standard et une traduction pour une plate-forme embarquée du type JavaCard. Nous commençons par détailler les spécificités de la plate-forme JavaCard. Ensuite nous montrons l'approche préconisée sur un exemple simple de porte-monnaie électronique. Nous spécifierons cet exemple en B puis nous donnerons la traduction pour une plate-forme Java standard. Nous représenterons cet exemple sous la forme d'une applet écrite manuellement pour une JavaCard. Enfin nous étudierons comment intégrer le porte-monnaie électronique spécifié en B en modifiant l'applet pour une plate-forme Java standard pour obtenir une plate-forme embarquée JavaCard.

## 13.2 La plate-forme JavaCard

### 13.2.1 Les spécificités de la plate-forme JavaCard

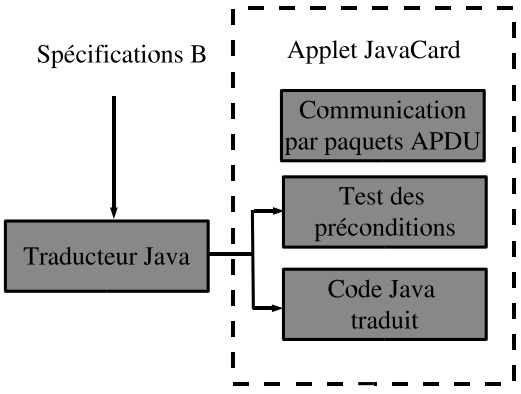


FIG. 13.1 – Développement d'une applet JavaCard

MACHINE
*BWallet*
CONSTANTS
*MAX_BALANCE, MAX_TRANSACTION_AMOUNT, DEFAULT_BALANCE*
PROPERTIES
$MAX\_BALANCE \in \text{NAT} \wedge MAX\_BALANCE < 50000 \wedge$
$MAX\_TRANSACTION\_AMOUNT \in \text{NAT} \wedge DEFAULT\_BALANCE \in \text{NAT} \wedge$
$DEFAULT\_BALANCE \leq MAX\_BALANCE$
CONCRETE_VARIABLES
*balance*
INVARIANT
$balance \in 0..MAX\_BALANCE$
INITIALISATION
$balance := DEFAULT\_BALANCE$
OPERATIONS
...
*setBalance (balanceInit)* $\widehat{=}$
*PRE*
$balanceInit \in NAT \wedge$
$balanceInit \in 0..MAX\_BALANCE$
*THEN*
$balance := balanceInit$
*END ;*
*debit (debitAmount)* $\widehat{=}$
*PRE*
$debitAmount \in NAT \wedge$
$(debitAmount \geq 0) \wedge$
$(debitAmount \leq MAX\_TRANSACTION\_AMOUNT) \wedge$
$(balance - debitAmount \geq 0)$
*THEN*
$balance := balance - debitAmount$
*END ;*
*credit (creditAmount)* $\widehat{=}$
PRE
$creditAmount \in \text{NAT} \wedge (creditAmount \geq 0 ) \wedge$
$(creditAmount \leq MAX\_TRANSACTION\_AMOUNT) \wedge$
$((balance + creditAmount) \leq MAX\_BALANCE)$
THEN
$balance := balance + creditAmount$
END
END

FIG. 13.2 – Spécification abstraite B du porte-monnaie électronique

IMPLEMENTATION
*BWallet_imp*
...
VALUES
*MAX_BALANCE = 10000 ;*
*MAX_TRANSACTION_AMOUNT = 100 ;*
*DEFAULT_BALANCE = 0*
...
END

FIG. 13.3 – Implémentation B du porte-monnaie électronique

Les spécificités de la plate-forme JavaCard [75, 12] par rapport à la plate-forme Java standard sont les suivantes :

- Il n'y a pas de chargement dynamique de classes.
- On peut créer dynamiquement des objets (new) en mémoire permanente mais il n'y a pas de ramasse-miette, ni de moyen pour désallouer explicitement les objets. La mémoire allouée ne peut donc être récupérée or la quantité de mémoire est faible. Il y a toutefois la possibilité de créer des objets transitoires (en RAM) en passant par l'API du système.
- Les types de base autorisés sont les **`byte`** (8 bits), les **`short`** (16 bits) et les **`boolean`**. Le type **`int`** (32 bits) peut être supporté de façon optionnelle.
- On ne dispose que des tableaux à une dimension.
- Il n'y a pas de threads.
- La politique de sécurité est incorporée dans la machine virtuelle.
- Les applications destinées à cette plate-forme sont des applets JavaCard : une des classes doit hériter de la classe ***`Applet`*** et implémenter au minimum les fonctions ***`install`*** (pour l'installation d'une applet) et ***`process`*** (pour traiter les commandes destinées à l'applet). Pour garantir un fonctionnement cohérent il y a un mécanisme de gestion des transactions (à un seul niveau). Un mécanisme de gestion de partage d'objets entre applets existe également.
- La communication entre la carte et le lecteur est réalisée par un protocole utilisant des paquets ADPU.

Il faut aussi remarquer que la plate-forme JavaCard est en évolution rapide et que des restrictions par rapport à la plate-forme Java sont levées au fur et à mesure des avancées technologiques et de la publication des nouvelles spécifications.

### 13.2.2 Structure d'une applet pour JavaCard

La structure des applets pour JavaCard est composée de trois parties (fig 13.1) :

- un module de communication,
- les préconditions des opérations traduites en Java,
- le code des implémentations en B0.

Le module de communication gère le flux de paquets APDU entre le lecteur et la carte à puce. Le protocole utilisé permet d'identifier l'applet qui communique par un identifiant.

```
IMPLEMENTATION
    UsingBWallet_imp
REFINES
    UsingBWallet
IMPORTS
    BWallet, BT_IO
OPERATIONS
    main ≙
    VAR balanceAmount, creditAmount, balanceFuture IN
        balanceAmount ← getBalance ;
        INT_WRITE (balanceAmount) ;
        creditAmount := 100 ;
        balanceFuture := balanceAmount + creditAmount ;
        IF (creditAmount ≤ 0) ∧ (creditAmount ≤ MAX_TRANSACTION_AMOUNT) ∧
            (balanceFuture ≤ MAX_BALANCE)
        THEN
            credit (creditAmount)
        END ;
        balanceAmount ← getBalance ;
        INT_WRITE (balanceAmount)
    END
END
```

FIG. 13.4 – La spécification B de l'exemple du porte-monnaie

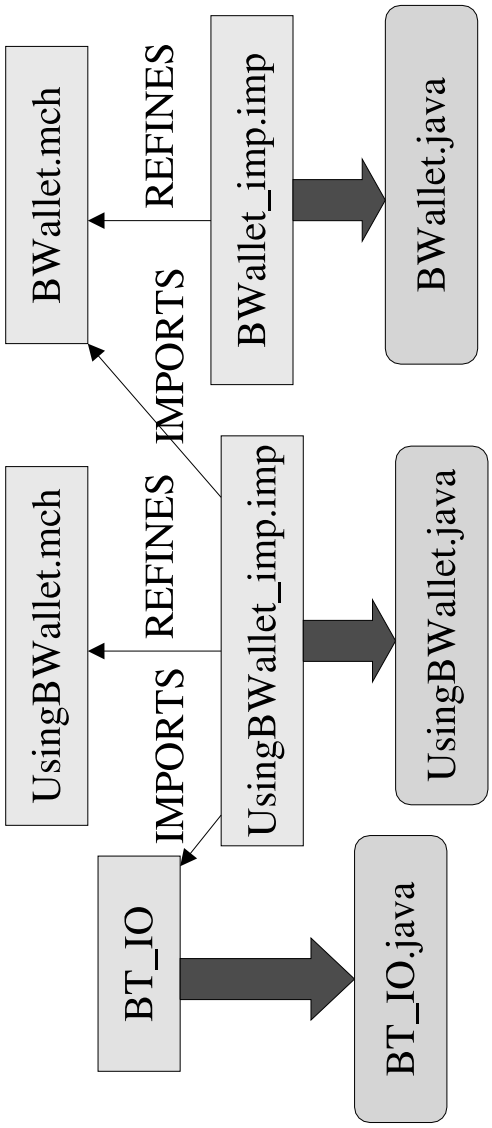


FIG. 13.5 – Traduction de l'applet *BWallet*

## 13.3 L'exemple du porte-monnaie électronique

### 13.3.1 Présentation

L'exemple du porte-monnaie électronique supporte les trois fonctionnalités suivantes : le débit, le crédit et la vérification du solde du porte-monnaie.

### 13.3.2 Les machines B

Le projet B (fig 13.2 et 13.4), entièrement prouvé, comprend une machine abstraite et son implémentation représentant le porte-monnaie électronique et une machine abstraite et son

```
/**** class BWallet.java * generated on 20 févr. 2003 19 :06 :28 ****/

package wallet;

class BWallet {
    static final int MAX_BALANCE = 10000;
    static final int MAX_TRANSACTION_AMOUNT = 100;
    static final int DEFAULT_BALANCE = 0;
    static int balance;

    static void INITIALISATION() {
        balance = 0;
    }
    public static void setBalance ( int balanceInit) {
        balance = balanceInit;
    }
    public static void debit ( int debitAmount) {
        balance = balance - debitAmount;
    }
    public static void credit (int creditAmount) {
        balance = balance + creditAmount;
    }
    public static int getBalance () {
        int amount = 0;
        amount = balance;
        return amount;
    }

}
```

FIG. 13.6 – Code Java généré pour l'exemple du porte-monnaie : contenu du porte-monnaie

implémentation permettant d'utiliser le porte-monnaie électronique. Nous ne présentons ici que la machine abstraite *BWallet* qui permet de voir les préconditions (l'implémentation ne faisant apparaître que la clause VALUES pour valuer les constantes) et l'implémentation qui utilise *BWallet* (*UsingBWallet_imp*).

### 13.3.3 La traduction en Java

La génération du code Java engendre autant de classes que d'implémentations B. Ces classes appartiennent toutes au package `wallet` représentant le projet B. Une implémentation en Java de la machine de base *BT_IO* (fig 13.5) utilisée est rajoutée automatiquement au projet. Il n'y a dans la spécification B qu'une instance du porte-monnaie électronique aussi l'optimisation est appliquée : le porte-monnaie est représenté par une classe dont les champs

et les méthodes sont statiques (fig 13.6 et 13.7). Cette stratégie est moins gourmande en mémoire que celle qui vise à utiliser une classe dont les méthodes et champs ne sont pas statiques.

## 13.4 Génération de code JavaCard

### 13.4.1 L'applet JavaCard du porte-monnaie électronique

Cette applet est une application JavaCard [1] classique développée dans [12]. Quel que soit le processus de développement utilisé manuel (fig 13.8) ou semi-automatique (fig 13.9) on est obligé de prendre en compte les spécificités de la JavaCard décrite en 13.2.1. On peut remarquer que `Wallet` hérite de la classe `Applet`. Une fois l'applet installée (l'environnement système appelé le Java Card Runtime Environment appelle la fonction `install`), celle-ci est sélectionnée puis doit traiter les commandes par la fonction `process`. Cette fonction reçoit des paquets (APDU) envoyés depuis le terminal dans lequel se trouve la carte et doit les interpréter pour récupérer les informations qui s'y trouvent. Seule l'opération de crédit est ici donnée.

### 13.4.2 Le processus de développement JavaCard semi-automatique

A partir du code de l'exemple 13.8 écrit manuellement, on peut voir que la spécification complète de telles applications serait particulièrement complexe à écrire en B (traitements de bas niveau avec des opérations de décalage de bits, lancement d'exception, communication avec l'extérieur). D'autre part une grande partie du code est liée à la technologie actuelle qui peut évoluer rapidement. C'est pourquoi nous proposons de réécrire l'application précédente en la découpant en deux classes :

- L'une qui est la logique même de l'application *BWallet* et qui pourra être spécifiée et prouvée en B (fig 13.2 et 13.4). Le traducteur Java permet de générer le code Java correspondant utilisable sur JavaCard (fig 13.6) depuis la spécification 13.2. Cependant le développement B devra respecter les règles édictées par le développement JavaCard :
  - entier court,
  - tableau à une dimension,
  - pas de retour multiples de paramètres pour éviter les créations dynamiques d'objets utilisés pour traduire les résultats multiples d'opération.
- L'autre, écrite manuellement, qui fait l'interface avec l'extérieur, la classe *Wallet*, et qui est le reflet de la technologie JavaCard du moment. Cette partie permettra l'encapsulation du protocole de communication dont le code est visible sur la figure 13.8 dans les méthodes `process` et `credit` (début de la méthode). On parle de processus de développement semi-automatique car il faudra vérifier manuellement que les fonctions spécifiées en B seront bien appelées après le test de leur précondition (fig 13.9).

Il faut noter ici que la personne qui génère le code Java doit déterminer si le porte-monnaie électronique devra être instancié plusieurs fois avant de souhaiter une quelconque optimisation. Par défaut et pour plus de sécurité, le générateur de code engendre du code Java permettant l'instanciation multiple.

[1] http ://java.sun.com/products/javacard/

```
/*** class UsingBWallet.java * generated on 20 févr. 2003 19 :06 :28 ****/

package wallet;

class UsingBWallet {

    static void INITIALISATION() {
        BWallet.INITIALISATION();
        BT_IO.INITIALISATION();
    }

    public static void main ( String Args[]) {
        INITIALISATION();
        int balanceAmount;
        int creditAmount;
        int balanceFuture;
        balanceAmount = BWallet.getBalance();
        BT_IO.writeInteger(balanceAmount);
        creditAmount = 100;
        balanceFuture = balanceAmount + creditAmount;
        if (creditAmount >= 0
            && creditAmount <= BWallet.MAX_TRANSACTION_AMOUNT
            && balanceFuture <= BWallet.MAX_BALANCE) {
            BWallet.credit(creditAmount);
        }
        balanceAmount = BWallet.getBalance();
        BT_IO.writeInteger(balanceAmount);

    }
}
```

FIG. 13.7 – Code Java généré pour l'exemple du porte-monnaie : point d'entrée du code Java

```
package wallet;
import javacard.framework.*;

public class Wallet extends Applet {
    ...
    final static byte WALLET = 0xAA;
    // codes of INS byte in the command APDU header
    final static byte GET = 0x01;
    final static byte CREDIT = 0x02;
    final static byte DEBIT = 0x03;
    final static short MAX_BALANCE = 10000;
    final static short MAX_TRANSACTION_AMOUNT = 100;
    private short balance;
    public Wallet() { balance = 0; register(); }
    public static void install( APDU apdu ) { new Wallet(); }
    public void process( APDU apdu ) {
        byte[] buffer = apdu.getBuffer();
        if ( buffer[ISO.OFFSET_CLA] != WALLET )
            ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
        switch (buffer[ ISO.OFFSET_INS]) {
            case GET : getBalance(buffer); break;
            case CREDIT : credit(buffer); break;
            case DEBIT : debit(buffer); break;
            default : ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
        }
    }
    ...
    void credit( Byte[] buffer) {
        byte byteRead = (byte) apdu.setIncomingAndReceive();
        if (byteRead != 1) ISOException. throwIt( ISO.SW_WRONG_LENGTH);
        short creditAmount = (buffer[ISO.OFFSET_CDATA]<< 8)
            | buffer[ISO.OFFSET_CDATA+1];
        // process data
        if ((creditAmount > MAX_TRANSACTION_AMOUNT)||(creditAmount < 0 ))
        ISOException.throwIt (INVALID_TRANSACTION_AMOUNT);
        // check the new balance
        if ((balance - creditAmount) < MAX_BALANCE)
            ISOException.throwIt(EXCEED_MAXIMUM_BALANCE);
        balance = balance - creditAmount;
    }
}
```

FIG. 13.8 – Le porte-monnaie électronique pour JavaCard écrit manuellement

```
package wallet;
import javacard.framework.*;
public class Wallet extends Applet {
    ...

    BWallet bw;
    public Wallet() { bw = new BWallet(); register(); }
    public static void install( APDU apdu ) { new Wallet(); }
    public void process( APDU apdu ) {
        ...
    }
    void credit(Byte[] buffer) {
        ...
        bw.credit(creditAmount);
    }
}
```

FIG. 13.9 – Implémentation JavaCard de la classe *Wallet*

# Chapitre 14

# Conclusion sur la génération de code objet à partir de spécifications B

## 14.1 Traduction de spécifications B en langage à objets

L'essentiel du développement du générateur de code Java est terminé. Il est diffusé sous une forme Open-Source. Pour garantir une chaîne de traduction ouverte et relativement complète, il est basé sur les JBTOOLS (cf chapitre 15) qui est une plate-forme sous licence GPL pour B développée au Laboratoire d'Informatique de l'Université de Franche-Comté (LIFC). Les principales caractéristiques de la chaîne de traduction sont les suivantes :

- Elle est, actuellement, basée sur la syntaxe du B et du B0 supporté par les ateliers commerciaux. Il existe néanmoins une restriction qui est l'interdiction d'instancier les machines prenant comme paramètres des ensembles par le type *STRING*.
- Elle supporte le multi-instance de machines B et est capable d'optimiser le code s'il n'y a qu'une seule instance.
- Elle permet le passage des tableaux en paramètre des fonctions par référence plutôt que par copie sous certaines contraintes. La preuve de la validité de cette possibilité est détaillée dans [80]

Nous avons défini une méthodologie d'écriture d'applet JavaCard pour permettre l'utilisation du traducteur Java. Cette traduction spécifique est plus contraignante du fait des spécificité de la plate-forme cible. En effet sur une plate-forme JavaCard on ne peut pas utiliser :

- les tableaux à plusieurs dimensions,
- les entiers longs,
- les opérations à résultats multiples.

D'autre part elle ne peut pas être complètement automatisée car le protocole de communication entre le lecteur et la carte est propre à la génération de JavaCard. De plus il est nécessaire de créer les méthode `process` et `install` dans la classe qui sert de point d'entrée de l'applet.

## 14.2 Evaluation

Le traducteur de spécifications B vers les langages à objets Java est Open-Source dans le but de permettre une diffusion des outils auprès de la communauté B. Il est optimisé en terme d'occupation mémoire et permet la réalisation en B d'applications prouvées pour les JavaCards. Un manuel utilisateur [101] et un guide d'extension du traducteur [100] ont été rédigés. Afin de montrer l'extensibilité du traducteur à d'autres langages à objet, le support du langage C# a été réalisé en peu de temps car les deux langages possèdent des caractéristiques similaires. En dépit des quelques différences syntaxiques mineures, il existe une différence de taille qui concerne les résultats de méthode. En effet le langage C# autorise plusieurs résultats au contraire de Java qui n'en accepte qu'un seul.

Afin de pouvoir réaliser des applications indépendamment des ateliers commerciaux des machines de base spécifiques au traducteur pour les entrées-sorties, les ensembles et les séquences ont été développées : les abstractions en B, les implémentations en Java et en C#.

Deux communications internationales [102, 97] et plusieurs présentations nationales [31], [95] au groupe B du GDR ALP ont été faites sur la plate-forme ouverte. Une démonstration a été acceptée au mois de Juin 2003 à la conférence internationale ZB2003 qui regroupe les communautés Z et B.

L'évaluation des traducteurs [94] s'est faite sur un ensemble d'exemples et de petits projets développés spécifiquement ou librement disponibles. Ces exemples sont livrés avec le traducteur. La génération du code a été effectuée et le code a été compilé et exécuté :

- en Java avec le kit de développement de Sun Microsystem,
- en C# sous Windows avec le compilateur de Microsoft,
- en C# sous Linux avec le compilateur go-mono [1].

Il faut aussi noter qu'un étudiant de l'université de Turku (Finlande) a testé le générateur de code et a permis ainsi de déceler plusieurs erreurs. Ceci ne nous apparaissant pas suffisant, nous avons repris l'exemple conséquent du DAB [2] (Distributeur Automatique de Billets). Sur cet exemple, les seules difficultés rencontrées ont été dues à la conversion des machines de base écrites initialement en C en Java. Les scénarios de tests décrits dans la documentation de cet exemple à partir de l'interface Tcl/Tk ont été passés avec succès. L'exemple du DAB est disponible sur le site WEB de BOM[3] avec l'accord pour les sources B et Tcl/Tk de ClearSy et de Fernando Mejia (Alsthom Transport Signalisation).

[1] http ://www.go-mono.org
[2] http ://www.atelierb.societe.com/DAB/DAB_HTML/FR/main.html
[3] http ://lifc.univ-fcomte.fr/~tatibouet/WEBBOM

# Quatrième partie

# Outils et applications

# Chapitre 15

# La plate-forme jBTools

## 15.1 Introduction

Le projet jBTools [1] [102, 31, 95, 98] est né du constat de l'absence pour la méthode B d'outils utilisables aussi bien à des fins d'enseignement que de développement de prototypes de recherche car les outils industriels ont un coût non négligeable, nécessitent l'accès à un serveur de licences et les sources ne sont évidemment pas accessibles. Le projet jBTools vise les objectifs suivants :

- La mise à disposition d'une plate-forme de base permettant aux équipes de recherche de développer des prototypes permettant la mise en évidence des idées.
- Cette plate-forme de base supporte un nombre suffisant de constructions du langage B pour rendre valide les expérimentations de recherche.
- Cette plate forme vise un ensemble aussi large que possible de systèmes matériels et logiciels (Windows, MacOs, Unix, ...) en économisant les ressources humaines. C'est dans cette optique qu'a été effectué le choix du langage Java.

Cette plate-forme et notamment le parser ont servi de base de construction pour de nombreux projets exploratoires :

- Gemplus : traduction [81] de spécifications formelles B vers C pour cartes à puces.
- LSR-IMAG :
  - La BoB [2] (Boîte à Outils B) est un ensemble de classes Java permettant de manipuler des constructions B, telles que des substitutions ou des prédicats, et de notamment de calculer des prédicats de *WP* (plus faible précondition).
  - GénéSyst[3] est un ensemble de classes utilisant la BoB pour générer des systèmes de transitions étiquetées représentant le comportement exact d'une spécification ou d'un raffinement B événementiel.
- ProB [4] : Notre outil permet d'utiliser ProB [67]. ProB est un outil de model-checking qui s'exécute sur les fichiers XML générés par la plate-forme.

[1] http ://lifc.univ-fcomte.fr/∼tatibouet/JBTOOLS
[2] http ://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/
[3] http ://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/
[4] http ://www.ecs.soton.ac.uk/∼mal/systems/prob.html

## 15.2 La plate-forme commune

Nous avons développé des outils permettant d'une part l'extraction de diagrammes UML (partie 1), et d'autre part la traduction vers les langages à objets tels que Java et C# (partie2). Le point de départ de ces deux processus est un projet B. Il est donc normal de définir une plate-forme commune pour factoriser le développement des premières phases d'analyse du projet B.

### 15.2.1 Architecture et principe de fonctionnement

La plate-forme (fig 15.1) est divisée en deux grandes parties qui sont l'analyseur (Parser et Typechecker) et la B Object Library (BOL). Les fichiers d'un projet B sont tout d'abord

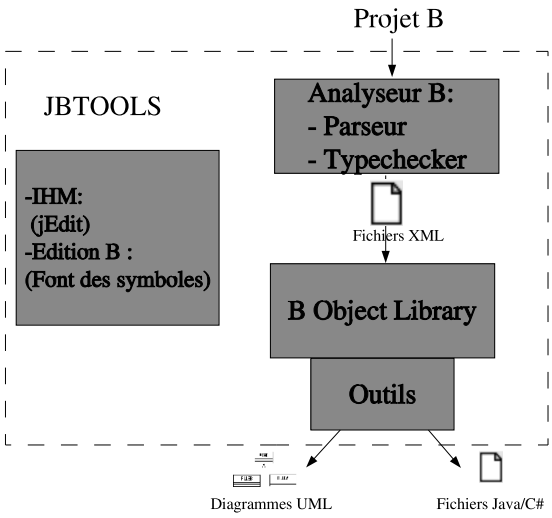


FIG. 15.1 – Architecture de la plate-forme JBTOOLS

analysés puis le typage de leurs entités est vérifié. Ils sont traduit en fichiers XML. Les fichiers résultats XML sont analysés pour obtenir un arbre abstrait à l'aide de la BOL. Chaque outil appartenant à la plate-forme JBTOOLS comporte un module dans la BOL.

### 15.2.2 Le TypeChecker

Le TypeChecker est divisé en deux sous parties : le parseur B et le typechecker lui-même. Le premier permet l'analyse syntaxique de la spécification B tandis que l'autre permet une analyse sémantique. Le principal but du Typechecker est de vérifier le typage. Ce dernier est enregistré dans une partie du fichier XML de sortie du typechecker appelé Contexte B. Les vérifications de type des entiers ont été limitées sur le type $\mathbb{Z}$ en ne tenant pas compte des dépassements maximum et minimum, ceci est réservé au prouveur. En effet, le typage des expressions entières ($\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, etc) est $\mathbb{P}(\mathbb{Z})$ (cf Tab 2.6) selon [93]. Dans ces conditions on ne cherchera pas à tester si un entier par exemple 100 est bien compris entre les bornes

minimales et maximales d'une variable *var* typée de la manière suivante var $\in$ 0..15. Selon le cycle de développement B (fig 2.1), la spécification est prouvée avant la traduction.

### 15.2.3 Le Contexte B

Le contexte B correspond à une mémorisation de l'existence et du type des entités locales à la machine et des entités des machines abstraites que la machine propriétaire du contexte rafine. Il est généré lors des deux phases d'analyse des composants B :

- lors de l'analyse syntaxique et sémantique,
- lors de la vérification du typage.

En réalité la majeure partie des entités est créée lors de la première phase. Ces entités sont les suivantes :

- ensembles énumérés ou valués,
- machines référencées et type de référencement,
- constantes abstraites et concrètes,
- variables abstraites et concrètes,
- opérations : arguments et résultats.

Certaines sont même typées à cet instant sachant que leur type peut être réévalué durant les opérations de recouvrement de type (cf 12.5.1.1). Durant la vérification de types le reste des entités est typé comme les variables locales définies par la substitution VAR ... IN ... END. Le contexte B permet dans un premier temps vérifier la cohérence du typage entre les différents composants de la chaîne de raffinement puisqu'il enregistre les types des entités des composants abstraits. Par exemple, on effectue un test sur la cohérence du typage des opérations entre les machines abstraites et les autres composants B de la chaîne de raffinement. Ensuite le contexte B aura pour fonction, durant la traduction en XML, d'exprimer le typage des entités.

### 15.2.4 Le format XML

Chaque fichier au format XML représente un composant B et comporte bien entendu le code de l'exécution des opérations mais aussi le contexte B nécessaire au typage des entités locales, externes (machine vues, importées, ... par les clauses SEES, IMPORTS, ...) et abstraites (composant raffiné par le composant considéré) de la machines (fig 15.2). Le début de la grammaire des fichiers XML générés par le TypeChecker est décrite dans la figure 15.2.

### 15.2.5 La traduction du typage en XML

Le traitement du typage est une fonctionnalité importante de l'outil pour la traduction en langages à objets du fait de la correspondance indirecte entre les types B et Objets (Java et C#). Les informations conservées pour le typage durant la traduction en XML sont définies dans la figure 15.3.

### 15.2.6 La B Object Library (BOL)

A chacune des balises XML (fig 15.2,15.3 et 15.4) correspond une classe de la B Object Library telle que les classes `Substitution`, `Parallel`(fig 15.5) et `Precondition` (fig 15.6).

Start : := "< xml version=1.0 standalone='yes' >" Bcomponent

BComponent : := Machine | Refinement | Implementation

BContext : :=
"<BContext>"
[AbstractBComponent] [AbstractParameters]
[AbstractEntities] [AbstractOperations]
[LocalParameters] [LocalEntities] [LocalOperations]
[ReferencedBComponents] [ExternEntities] [ExternOperations]
"</BContext>"

FIG. 15.2 – Début de la grammaire XML

Type : :=
TypeIdentifier
| TypeString
| TypeBool
| TypeEntier
| TypeParties
| TypeProduitCartesien

TypeIdentifier : := "<TypeIdentifier>"IDENTIFIER"</TypeIdentifier>"
TypeString : := "<TypeString>""</TypeString>"
TypeBool : := "<TypeBool>""</TypeBool>"
TypeEntier : :="<TypeInteger>""</TypeInteger>"
TypeParties : :="<TypePowerSet>"Type"</TypePowerSet>"
TypeProduitCartesien : := "<TypeCartesianProduct>"Type Type "</TypeCartesianProduct>"

FIG. 15.3 – Traduction du typage en XML

La classe *substitution* est une classe mère dont les classes *Parallel*, *Sequencing* et *Precondition* vont hériter (fig 15.6). Ceci est décrit dans la grammaire par la ligne de définition des substitutions :
Substitution : := Parallel | Sequencing | Precondition | ...

La BOL permet la création d'un arbre de syntaxe concrète XML qui correspond à chaque fichier XML. La traduction B vers XML a été validée par la décompilation XML vers B. Ainsi en effectuant le cycle itératif complet (B vers XML vers B), on s'assure partiellement que le schéma de traduction est correct et complet. La BOL est à la base des différents outils d'extraction de diagrammes UML et de traduction vers les langages à objets du type Java et C#. En effet, il suffit d'ajouter une méthode d'extraction ou de traduction dans chaque classe de la BOL, selon l'outil considéré, dans l'optique d'une exécution récursive de la racine (**classe** *Start*, correspondant à la grammaire de la figure 15.2) aux feuilles de l'arbre (fig

Machine : :=
"<Machine>" BContext Header [Constraints][Includes][Uses]
[Sees][Sets][ConcreteConstants]
[AbstractConstants][Properties][ConcreteVariables]
[AbstractVariables][Invariant][Assertions]
[Initialization][Operations]
"</Machine>"

Substitution : := Parallel | Sequencing | Precondition | ...
Parallel : := "<Parallel>" Substitution (Substitution)+ "</Parallel>"
Sequencing : := "<Sequencing>" Substitution (Substitution)+ "</Sequencing>"
Precondition : := "<Precondition>" Predicate Substitution "<Precondition>"

FIG. 15.4 – Une partie de la grammaire des fichiers XML

```
abstract class Substitution extends BObject {}

class Parallel extends Substitution {
    Substitution[] substitutions;
    public Parallel() {}
    public void setFields (BObject[] values) {
        substitutions = new Substitution[values.length];
        for (int i = 0; i < values.length; i += 1 ) {
            substitutions[i] = ((Substitution) values[i]);
        }
    }
}
```

FIG. 15.5 – Exemple de classes Java de la BOL correspondant aux substitutions *parallèles*

15.5 et 15.6).

## 15.3 Les outils

Dans les paragraphes précédents nous avons présenté le projet jBTools qui comprend un analyseur syntaxique et sémantique pour la notation B et un vérifieur de types. Le vérifieur de types part des arbres produits par l'analyse syntaxique d'un ensemble de fichiers B pour effectuer un contrôle du typage et produire un ensemble de fichiers XML dans lesquels on retrouve les éléments du source B mais aussi les types. A ce format XML correspond une bibliothèque de classes utilisée par les outils lors de la lecture d'un fichier XML pour créer une arborescence d'objets représentant le contenu du fichier. La plate-forme jBTools possède aussi un plug-in pour l'éditeur jEdit de texte et de code (fig 15.7) qui permet :

– la coloration syntaxique,

```
class Precondition extends Substitution {
    Predicate precondition;
    Substitution substitution;
    public Precondition() {}
    public void setFields (BObject[] values) {
        precondition = (Predicate) values[0];
        substitution = (Substitution) values[1];
    }
}
```

FIG. 15.6 – Autre exemple de classes Java de la BOL correspondant aux substitutions *préconditions*

- une aide à base d'icônes pour les symboles mathématiques utilisés dans la notation B,
- l'édition de plusieurs fichiers simultanément,
- l'appel du vérifieur de types et l'affichage des erreurs,
- la tradution en diagrammes UML (diagrammes de classes),
- la traduction complète d'un projet B complet en HTML avec les liens hypertextes correspondant aux liens sur les machines vues, importées etc.(clauses SEES, IMPORTS etc),
- la traduction en Java.

### 15.3.1 L'outil B2UML

B2UML est un des outils en cours de réalisation (fig 15.8). Son but est de supporter la démarche de ré-ingénierie de B vers UML que nous proposons en permettant d'appliquer les règles méthodologiques à un nombre suffisant d'exemples et ainsi valider notre démarche (partie 1). Il permet actuellement d'extraire les diagrammes de classes et certaines contraintes OCL. Il part des fichiers XML (un fichier par machine) pour produire un résultat qui est exploitable par l'outil libre ArgoUML [5] version 0.15.4. Ce résultat est composé de fichiers XMI (XML METADATA INTERCHANGE FORMAT) contenant les informations relatives à la modélisation UML et du projet ArgoUML propre à ce logiciel. Nous ne produisons pas les fichiers destinés au rendu visuel des diagrammes. L'automatisation n'est pas complète : ArgoUML dans sa version actuelle ne disposant pas d'un module permettant de redessiner automatiquement le diagramme de classes, il est nécessaire d'intervenir manuellement en sélectionnant les classes puis en les introduisant sur la zone de visualisation. Les associations sont redessinées à partir du moment ou les classes sont présentes.

Le choix d'ArgoUML se justifie par la possibilité d'intervenir si nécessaire sur le source, par la volonté des développeurs de suivre les normes les plus récentes (UML 2.0) et surtout pour sa gestion des contraintes OCL. En contrepartie, il reste malheureusement quelques inconvénients :

- ArgoUML est très orienté Java y compris dans sa visualisation des attributs et opérations qui diffèrent de celle d'UML,
- c'est un outil qui ne dispose pas de tous les types de diagrammes,

[5] http ://www.argouml.org

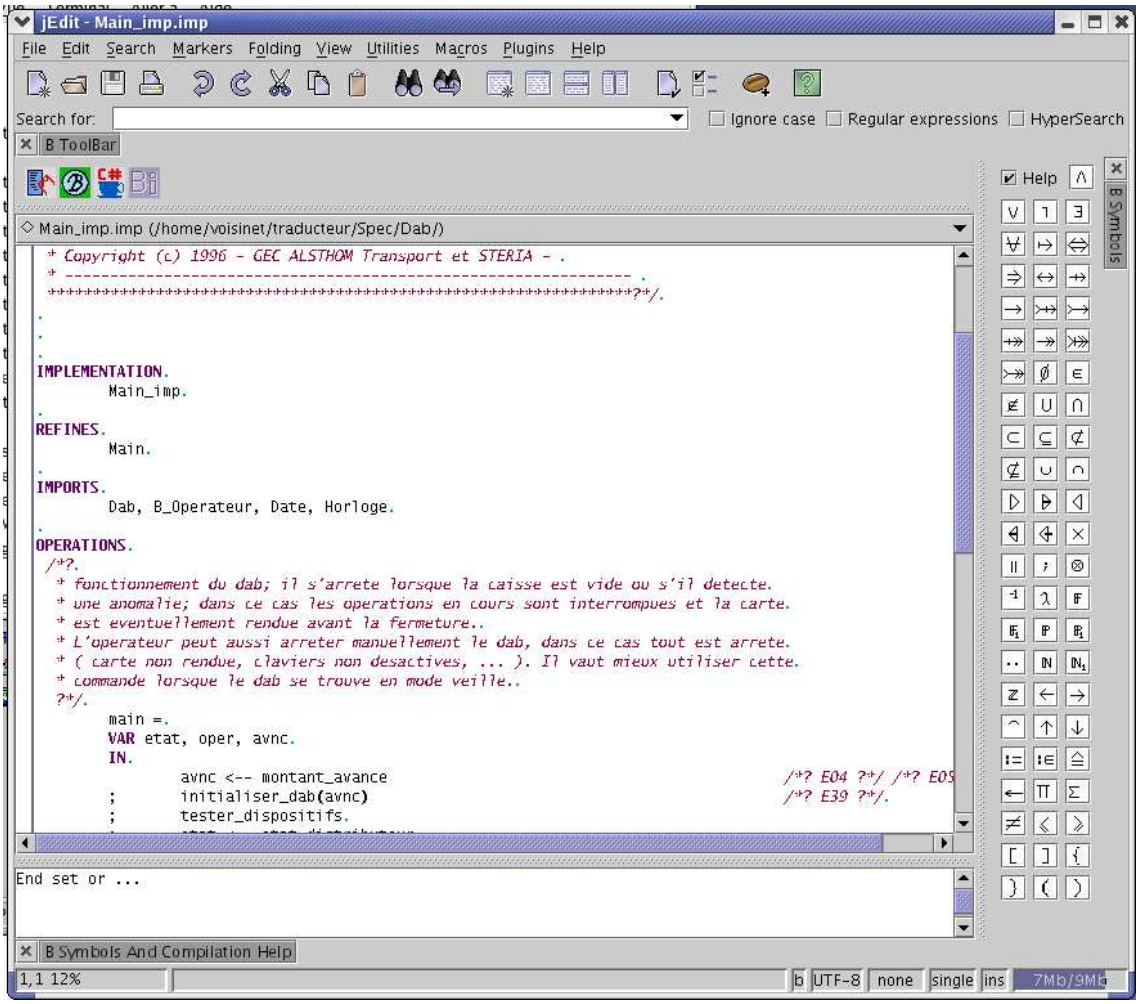


FIG. 15.7 – Le plugin pour jEdit

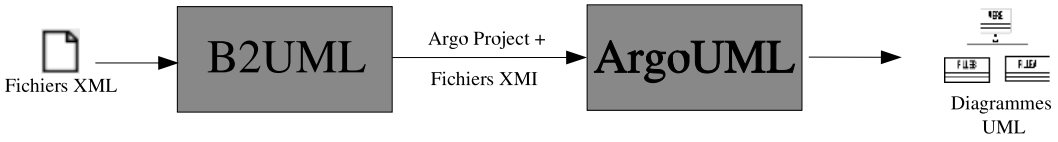


FIG. 15.8 – Architecture des différents outils

– sa gestion des contraintes OCL n'est pas complète.

#### 15.3.1.1 Un exemple

L'objectif de cet exemple très simplifié est la gestion de l'inscription d'étudiants à un diplôme. L'inscription ne suffit pas et il faut aussi que les étudiants s'inscrivent aussi aux

examens. La spécification B de l'exemple est donnée figure 6.26.

Les ensembles (clause SETS) *ETUDIANTS* et *DIPLOMES* permettent de représenter tous les étudiants et tous les diplômes potentiels. Les variables (clause VARIABLES) *Inscrits* et *InscritsAuxExamens* sont typées dans l'invariant (clause INVARIANT). *Inscrits* est une fonction partielle entre l'ensemble des étudiants et celui des diplômes : c'est donc un ensemble de couples ou le premier élément d'un couple est un étudiant et le deuxième un diplôme. *InscritsAuxExamens* est un sous ensemble de *Inscrits*.

Initialement aucun étudiant n'est inscrit. L'invariant est devenu vrai après l'initialisation (clause INITIALISATION). Dans la clause OPERATIONS on trouve l'ensemble des opérations. La seule opération décrite ici est Inscrire qui correspond à l'inscription d'un étudiant en vue d'obtenir un diplôme. Cette opération est soumise à une précondition qui doit être vérifiée. Cette précondition permet d'attribuer un type aux paramètres puis de vérifier que l'étudiant à inscrire n'est pas déjà inscrit.

Les opérations dans le langage B sont exprimées avec des substitutions généralisées comme :=, ||, ou PRE. Cette notion " opérationnelle " permet d'exprimer ce que font les opérations. De plus les substitutions peuvent être manipulées par le prouveur pour vérifier que l'invariant reste vrai après " l'exécution " de l'opération. Le corps de l'opération permet de rajouter le couple *(etudiant* $\mapsto$ *diplome)* à l'ensemble *Inscrits*.

#### 15.3.1.2 Les résultats obtenus avec B2UML

A partir de la spécification 6.26, l'outil B2UML produit une représentation qui est destinée à être relue plus spécifiquement par l'outil de modélisation ArgoUML mais à priori les fichiers XMI qui contiennent le modèle UML devraient pouvoir être utilisés à partir d'autres outils. Actuellement B2UML travaille sur une machine abstraite B et permet d'obtenir un diagramme de classes comme celui de la figure 15.9.

Sur ce diagramme une partie des règles adoptées apparaît :

- la machine est une classe ;
- un ensemble abstrait comme *ETUDIANT* est représenté par une classe *Elt_ETUDIANTS* correspondant au type d'éléments de l'ensemble et par une association entre cette classe et la classe représentant la machine. L'extrémité de l'association du côté de la classe correspondant au type d'éléments possède un nom (rôle) qui est celui de l'ensemble (ici ETUDIANTS). Les variables de type ensemble deviennent des associations avec leur nom comme nom de rôle du côté correspondant au type d'éléments ;
- la fonction partielle Inscrits est traduite par une classe *Elt_Inscrits* reliée par une association dont le rôle est *Inscrits* à la classe représentant la machine. La classe *Elt_Inscrits* est également reliée par une agrégation à la classe *Elt_ETUDIANTS* et par une autre à la classe *Elt_DIPLOMES* ;
- le processus est le même pour la fonction partielle *InscritsAuxExamens*. Pour indiquer que *InscritsAuxExamens* est un sous ensemble de *Inscrits*, on utilise soit une dépendance prédéfinie comme {**subset**} entre les deux associations, soit une contrainte OCL.

En fait ceci ne suffit pas pour décrire une fonction partielle comme *Inscrits* :

- une fonction est un ensemble de couples qui ne contient pas deux couples distincts ayant le même premier élément ;
- une fonction est partielle lorsque tous les éléments de l'ensemble de départ (*ETUDIANTS*) ne sont pas utilisés pour faire des couples.

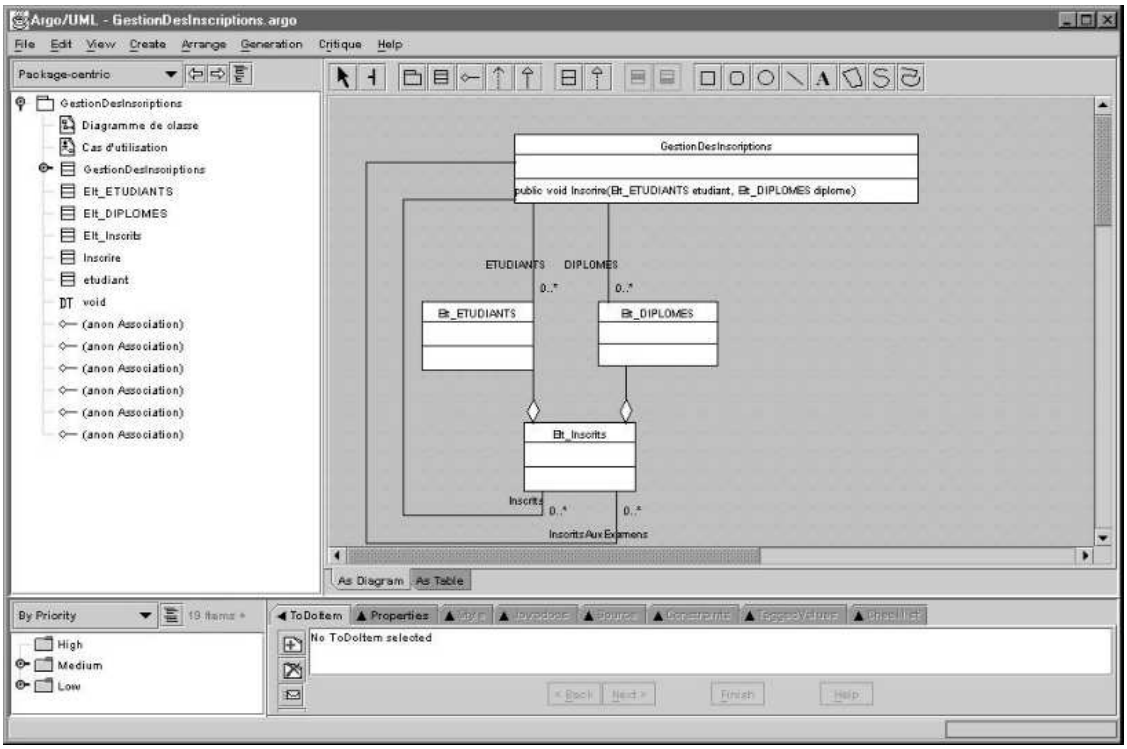


FIG. 15.9 – Diagramme de classes de la gestion des inscriptions

### 15.3.2 Traducteur vers Java/C#

Le traducteur vers les langages à objets est connecté sur la BOL comme l'outil B2UML. Les fichiers XML sont lus et l'arbre abstrait est généré à l'aide de la BOL. Les fichiers Java ou C# sont générés sachant que l'utilisateur définit les options suivantes :

- le type de compilation application/bibliothèque,
- le point d'entrée du projet,
- le type de base à utiliser.

Le type de compilation permet de créer des fichiers Java/C# avec ou sans point d'entrée. Le point d'entrée du projet est la machine principale qui va instancier les autres composants du projet B. Le lanceur de l'application tentera de l'exécuter. S'il existe, la traduction fera apparaître une application et dans le cas contraire ce sera une bibliothèque. Suite aux problèmes de recouvrement de types (cf 12.5.1.1) nous avons choisi de laisser le développeur choisir le type de base de l'application entre **`int`** et **`short`**, pour optimiser l'espace mémoire occupé, sachant qu'il faut absolument paramètrer ce type (par les bornes MININT et MAXINT) avant l'obtention de la preuve sans quoi elle n'a aucune consistance.

## 15.4 Conclusion

Dans ce chapitre nous avons présenté la plate-forme jBTools. Elle est composée de fonctionnalités communes d'analyse syntaxique, sémantique et de vérification de typage. Le résultat est enregistré dans des fichiers XML (un par composant B). Il est ensuite exploitable par la BOL sur laquelle sont branchés les principaux outils de la plate-forme qui sont :

- un traducteur vers les langage à objets,
- un extracteur de diagrammes UML.

Actuellement nous disposons d'un extracteur de diagrammes de classes et de contraintes OCL partiellement achevé et d'un traducteur finalisé. Nous pensons étendre le domaine d'application de l'outil B2UML d'une part en terminant les fonctionnalités de traduction vers les contraintes OCL et en y ajoutant les modules nécessaires aux extractions d'autres diagrammes. D'autre part notre plate-forme permet d'utiliser l'outil ProB [6]. ProB [67] est un outil complémentaire à ceux développés au sein du LIFC, de model-checking qui utilise les fichiers XML générés par notre plate-forme.

[6] http ://www.ecs.soton.ac.uk/~mal/systems/prob.html

# Chapitre 16

# Application et mise en oeuvre

## 16.1 Introduction

Dans ce chapitre il s'agit de mettre en application les différentes démarches définies dans ce travail de thèse sur un cas d'étude. Le cas d'étude considéré est un carrefour avec plusieurs feux tricolores de circulation. Nous allons appliquer les contributions de nos travaux sur ce projet B. Dans une première partie nous extrairons les diagrammes UML et contraintes OCL puis dans une deuxième partie nous allons exposer le résultat de la traduction du projet en une application exécutable.

## 16.2 Cahier des charges

Dans cette section, nous allons détailler les différentes règles de gestion du cahier des charges d'un carrefour possédant deux feux de circulation tricolores.

### 16.2.1 Présentation

Le carrefour possède deux feux tricolores. Nous avons choisi de ne pas prendre en compte la gestion des piétons pour se concentrer uniquement sur la circulation des automobiles. D'autre part nous ne traitons ni les temporisations (temps d'allumage des feux) ni la synchronisation du passage au rouge. En effet, les feux d'un carrefour sont à un instant tous au rouge puis ensuite l'un passe au vert. Dans notre modèle un feu passe au rouge simultanément au passage au vert de l'autre.

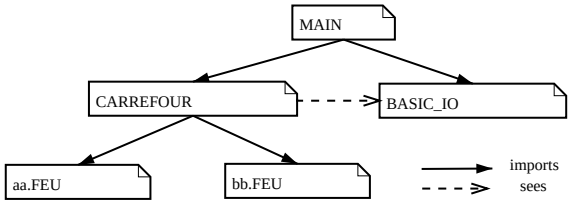


FIG. 16.1 – Graphe de dépendance du projet de carrefour

### 16.2.2 Propriétés du système

Les règles de gestion que doit respecter le système sont les suivantes :

1. un feu est muni de trois lampes rouge, orange et vert,
2. un feu tricolore ne peut avoir qu'une et une seule lampe à l'état allumé,
3. lorsque le système n'est pas en état de fonctionner tous les feux du carrefour passent à l'orange clignotant,
4. un feu peut être au vert ou à l'orange si et seulement si l'autre est au rouge.

## 16.3 Spécifications du projet B

Dans ce projet il existe trois composant distincts :
- le feu,
- le carrefour,
- le composant principal d'exécution.

Le programme principal qui contrôle le système complet importe le composant *CARREFOUR* qui lui-même importe deux instances du composant *Feu* (fig 16.1). Le feu est muni des trois lampes rouge, orange et vert.

### 16.3.1 Le composant *Feu*

Le composant *Feu* (fig 16.2, 16.3) gère le passage des lampes de l'état éteint (*off*) à allumé (*on*) sachant que deux lampes du feu tricolore ne peuvent pas être allumées en même temps.

MACHINE *Feu*
SETS $ETAT = \{on, off\}$
CONCRETE_VARIABLES *rr, oo, vv*
INVARIANT
$rr \in ETAT \wedge oo \in ETAT \wedge vv \in ETAT \wedge (rr = on \Leftrightarrow (oo = off \wedge vv = off)) \wedge$
$(oo = on \Leftrightarrow (rr = off \wedge vv = off)) \wedge (vv = on \Leftrightarrow (rr = off \wedge oo = off))$
INITIALISATION *rr, oo, vv := off, on, off*
OPERATIONS
*estRouge* $\leftarrow$ *estIlRouge* $\widehat{=}$ BEGIN *estRouge := bool (rr = on)* END;
*estOrange* $\leftarrow$ *estIlOrange* $\widehat{=}$ BEGIN *estOrange := bool (oo = on)* END;
*estVert* $\leftarrow$ *estIlVert* $\widehat{=}$ BEGIN *estVert := bool (vv = on) END;*
*devientRouge* $\widehat{=}$ BEGIN *rr, oo, vv := on, off, off* END;
*devientOrange* $\widehat{=}$ BEGIN *rr, oo, vv := off, on, off* END;
*devientVert* $\widehat{=}$ BEGIN *rr, oo, vv := off, off, on* END;
END

FIG. 16.2 – Spécification abstraite du composant *Feu*

IMPLEMENTATION *Feu_imp*
REFINES *Feu*
INITIALISATION *rr := off ; oo := on ; vv := off*
OPERATIONS
*estRouge* ← *estIlRouge* ≙ BEGIN *estRouge := bool (rr = on)* END ;
*estOrange* ← *estIlOrange* ≙ BEGIN *estOrange := bool (oo = on)* END ;
*estVert* ← *estIlVert* ≙ BEGIN *estVert := bool (vv = on)* END ;
*devientRouge* ≙ BEGIN *rr := on ; oo := off ; vv := off* END ;
*devientOrange* ≙ BEGIN *rr := off ; oo := on ; vv := off* END ;
*devientVert* ≙ BEGIN *rr := off ; oo := off ; vv := on* END ;
END

FIG. 16.3 – Implémentation du composant *Feu*

### 16.3.2 Le composant *CARREFOUR*

Le composant *CARREFOUR* (fig 16.4 et 16.5) gère les changements simultanés de couleurs des feux sachant que les deux feux ne peuvent pas permettre aux véhicules de passer en même temps sauf dans le cas particulier de l'orange clignotant lorsque le système n'est pas en état de fonctionner.

MACHINE *CARREFOUR*
SETS *COUL*={*rouge, orange, vert*}
ABSTRACT_CONSTANTS *suiv*
PROPERTIES
$suiv \in COUL \rightarrowtail\!\!\!\!\!\rightarrow COUL \wedge suiv(rouge) = vert \wedge$
$suiv(vert) = orange \wedge suiv(orange) = rouge$
VARIABLES *feu1, feu2*
INVARIANT
$feu1 \in COUL \wedge feu2 \in COUL \wedge ((feu1=orange \wedge feu2=orange)$ *or*
$(feu1=rouge \wedge feu2 \neq rouge)$ *or* $(feu1 \neq rouge \wedge feu2=rouge))$
INITIALISATION *feu1, feu2 := orange, orange*
OPERATIONS
*MiseEnService* ≙ SELECT $feu1 = orange \wedge feu2 = orange$ THEN *feu1, feu2 := rouge, vert* END ;
*ChangerFeux* ≙
SELECT $feu1 = vert \wedge feu2 = rouge$ THEN *feu1 := orange*
WHEN $feu1 = rouge \wedge feu2 = vert$ THEN *feu2 := orange*
WHEN $feu1 = rouge \wedge feu2 = orange$ THEN *feu1, feu2 := vert, rouge*
WHEN $feu1 = orange \wedge feu2 = rouge$ THEN *feu1, feu2 := rouge, vert*
END
END

FIG. 16.4 – Spécification abstraite du composant *CARREFOUR*

```
IMPLEMENTATION CARREFOUR_imp
REFINES CARREFOUR
SEES BASIC_IO
IMPORTS aa.Feu, bb.Feu
INVARIANT
   (aa.vv = on ⇔ (feu1 = vert) ) ∧ (aa.oo = on ⇔ (feu1 = orange)) ∧
   (aa.rr = on ⇔ (feu1 = rouge)) ∧ (bb.vv = on ⇔ (feu2 = vert)) ∧
   (bb.oo = on ⇔ (feu2 = orange)) ∧ (bb.rr = on ⇔ (feu2 = rouge))
INITIALISATION aa.devientOrange ; bb.devientOrange
OPERATIONS
   MiseEnService ≙
   VAR aaOrange, bbRouge IN
      aaOrange ← aa.estIlOrange ; bbOrange ← bb.estIlOrange
      IF aaOrange = TRUE ∧ bbOrange = TRUE THEN aa.devientRouge ; bb.devientVert END
   END;
   ChangerFeux ≙
      VAR aaRouge, aaVert, aaOrange, bbRouge, bbVert, bbOrange IN
      aaRouge ← aa.estIlRouge ; aaVert ← aa.estIlVert ;
      aaOrange ← aa.estIlOrange ; bbRouge ← bb.estIlRouge ;
      bbVert ← bb.estIlVert ; bbOrange ← bb.estIlOrange ;
      IF aaVert = TRUE ∧ bbRouge = TRUE THEN aa.devientOrange
      ELSIF aaRouge = TRUE ∧ bbVert = TRUE THEN bb.devientOrange
      ELSIF aaRouge = TRUE ∧ bbOrange = TRUE THEN aa.devientVert ; bb.devientRouge
      ELSIF aaOrange = TRUE ∧ bbRouge = TRUE THEN aa.devientRouge ; bb.devientVert
      END;
      aaRouge ← aa.estIlRouge ;
      IF aaRouge = TRUE THEN STRING_WRITE ("feu 1 Rouge") END;
      aaVert ← aa.estIlVert ;
      IF aaVert = TRUE THEN STRING_WRITE ("feu 1 Vert") END;
      aaOrange ← aa.estIlOrange ;
      IF aaOrange = TRUE THEN STRING_WRITE ("feu 1 Orange") END;
      bbRouge ← bb.estIlRouge ;
      IF bbRouge = TRUE THEN STRING_WRITE (", feu 2 Rouge\n") END;
      bbVert ← bb.estIlVert ;
      IF bbVert = TRUE THEN STRING_WRITE (", feu 2 Vert\n") END;
      bbOrange ← bb.estIlOrange ;
      IF bbOrange = TRUE THEN STRING_WRITE (", feu 2 Orange\n") END
   END
END
```

FIG. 16.5 – Implémentation du composant *CARREFOUR*

### 16.3.3 Composant principal

Le composant principal (fig 16.6) gère la boucle qui fait appel au changement de couleurs du composant *CARREFOUR*. Dans le but de prouver la convergence de la spécification, le

nombre d'itérations de la boucle est limité à 20 mais il doit être infini dans des conditions réelles de fonctionnement.

```
MACHINE MAIN
OPERATIONS
   main ≙ BEGIN skip END
END
```

```
IMPLEMENTATION MAIN_imp
REFINES MAIN
IMPORTS CARREFOUR, BASIC_IO
OPERATIONS
   main ≙
      VAR ii IN
      STRING_WRITE("Deux feux a un carrefour\n") ;
      STRING_WRITE("Mise en service des feux\n") ;
      MiseEnService ; ii := 20 ;
      WHILE ii > 0 DO
         STRING_WRITE ("Changer les feux\n") ;
         ChangerFeux ;
         ii := ii - 1
      INVARIANT ii ∈ NAT
      VARIANT ii
      END
   END
END
```

FIG. 16.6 – Le composant principal

## 16.4 Extraction de vues UML

Dans cette section nous avons extrait tous les diagrammes dont nous avons donné le principe dans ce document de thèse. Il ne s'agit pas d'extraire systématiquement tous les diagrammes possibles pour chaque spécification mais de conserver les diagrammes susceptibles d'aider la compréhension et la validation du système relativement au cahier des charges.

### 16.4.1 Diagramme de classes

Les deux spécifications significatives du projet B en terme d'invariants et d'opérations au niveau abstrait sont les spécifications des composants *CARREFOUR* et *Feu*. Nous avons donc extrait de celles-ci les diagrammes de classes respectifs des figures 16.7 et 16.9 depuis les prédicats de typage de l'invariant. Le diagramme de classes extrait de la spécification *CARREFOUR* doit être complété par la contraintes OCL (fig 16.8) qui permet d'exprimer les contraintes associées à la bijection totale *suiv*.

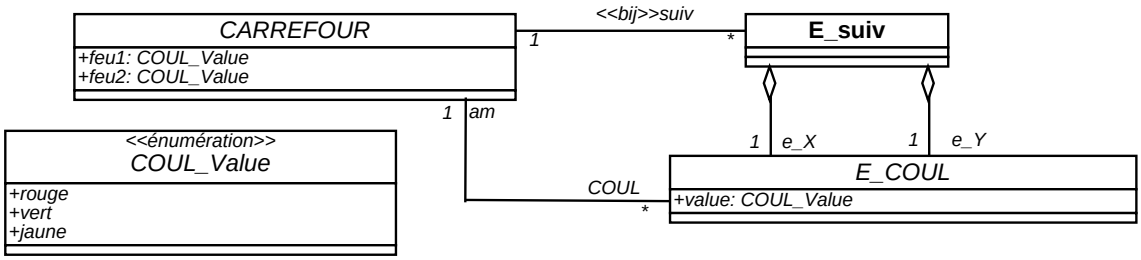


FIG. 16.7 – Diagramme de classes extrait de la spécification *CARREFOUR*

**context** *CARREFOUR* **inv :**
*suiv* → **forAll(***c1,c2 : E_suiv* | *c1 <> c2* **implies** *c1.e_X <> c2.e_X***)**
**and** *suiv*→ **forAll (***c1, c2 : E_suiv* | *c1.e_X <> c2.e_X* **implies**
*c1.e_Y <> c2.e_Y***)**
**and** *F* → **collect (***c : E_suiv* | *c.e_Y***)** → **includesAll(***E_COUL***)**
**and** *suiv*→ **size()**= *E_COUL* → **size()**

FIG. 16.8 – Contrainte relative à la bijection totale *suiv*

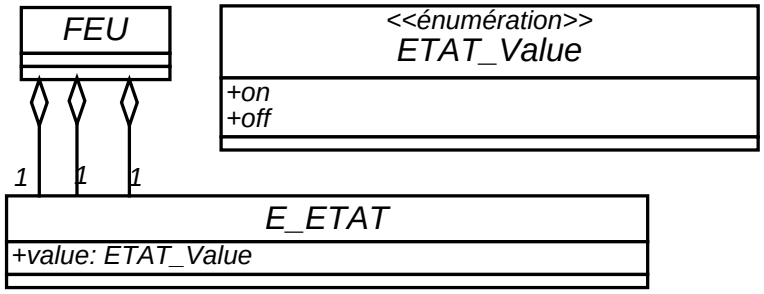


FIG. 16.9 – Diagramme de classes extrait de la spécification *Feu*

### 16.4.2 Contraintes OCL

Les prédicats de typage sont utilisés pour générer les diagrammes de classes. Les propriétés complexes de l'invariant sont réécrites en contraintes OCL associées aux diagrammes de classes. De plus les post-conditions des opérations sont aussi traduites en contraintes OCL.

#### 16.4.2.1 Les propriétés invariantes

Dans la spécification *CARREFOUR*, les prédicats définissant les restrictions des valeurs associées aux variables *feuA* et *feuB* sont traduits en la contrainte OCL présentée figure 16.10. Cette contrainte correspond à la règle de gestion 3 du cahier des charges.

Les propriétés complexes de l'invariant de la spécification *Feu* correspondent à la règle de gestion 2 du cahier des charges. Elles définissent les restrictions des valeurs associées des variables *rr, oo* et *vv*. L'équivalence ne peut être traduite directement en OCL c'est pourquoi on utilise deux implications pour générer la contrainte de la figure 16.11.

**context** *CARREFOUR* **inv :**
*feuA = COUL_Value : :orange* **and** *feuB = COUL_Value : :orange* **and**
*feuA = COUL_Value : :rouge* **and not**(*feuB = COUL_Value : :rouge)* **and**
**not**(*feuA = COUL_Value : :rouge)* **and** *feuB = COUL_Value : :rouge*

FIG. 16.10 – Contrainte relative aux propriétés complexes de l'invariant de la spécification *CARREFOUR*

**context** *Feu* **inv :**
(*rr = ETAT_Value : :on* **implies** (*oo = ETAT_Value : :off* **and** *vv = ETAT_Value : :off*)) **and**
((*oo = ETAT_Value : :off* **and** *vv = ETAT_Value : :off*) **implies** *rr = ETAT_Value : :on*) **and**
(*oo = ETAT_Value : :on* **implies** (*rr = ETAT_Value : :off* **and** *vv = ETAT_Value : :off*)) **and**
((*rr = ETAT_Value : :off* **and** *vv = ETAT_Value : :off*) **implies** *oo = ETAT_Value : :on*) **and**
(*vv = ETAT_Value : :on* **implies** (*rr = ETAT_Value : :off* **and** *oo = ETAT_Value : :off*)) **and**
((*rr = ETAT_Value : :off* **and** *oo = ETAT_Value : :off*) **implies** *vv = ETAT_Value : :on*)

FIG. 16.11 – Contrainte relative aux propriétés complexes de l'invariant de la spécification *Feu*

#### 16.4.2.2 Les post-conditions des opérations

Dans ce projet, il est intéressant de réécrire en OCL les post-conditions des opérations *MiseEnService* et *ChangerFeux* du composant *CARREFOUR* (fig 16.12 et 16.13) car ce composant comporte l'invariant le plus développé.

### 16.4.3 Diagramme d'états-transitions

Les diagrammes d'états-transitions à extraire sont ceux provenant des spécifications *Feu* et *CARREFOUR*. Dans la première spécification, *Feu*, afin d'alléger le diagramme figure 16.14 nous avons remplacé les valeurs *on* et *off* respectivement par *O* et *F*.

prd(SELECT *feu1 = orange* $\wedge$ *feu2 = orange* THEN *feu1, feu2 := rouge, vert* END)
$\Leftrightarrow$ prd((feu1 = orange $\wedge$ feu2 = orange) $\Longrightarrow$ feu1, feu2 := rouge, vert)
$\Leftrightarrow$ ((feu1 = orange $\wedge$ feu2 = orange) $\wedge$ prd(feu1 := rouge $\|$ feu2 := vert))
$\Leftrightarrow$ ((feu1 = orange $\wedge$ feu2 = orange) $\wedge$ (feu1' = rouge $\wedge$ feu2' = vert))

La contrainte OCL se déduit de cette post-condition :
**Context** *CARREFOUR : :MiseEnService()* **post :**
*((feu1@pre = COUL_Value : :orange* **and** *feu2@pre = COUL_Value : :orange)* **and**
(*feu1 = COUL_Value : :rouge* **and** *feu2 = COUL_Value : :vert*))

FIG. 16.12 – Extraction de la post-condition de l'opération *MiseEnService*

prd(SELECT *feu1 = vert* $\wedge$ *feu2 = rouge* THEN *feu1 := orange*
WHEN *feu1 = rouge* $\wedge$ *feu2 = vert* THEN *feu2 := orange*
WHEN *feu1 = rouge* $\wedge$ *feu2 = orange* THEN *feu1, feu2 := vert, rouge*
WHEN *feu1 = orange* $\wedge$ *feu2 = rouge* THEN *feu1, feu2 := rouge, vert*
END)
$\Leftrightarrow$
prd((feu1 = vert $\wedge$ feu2 = rouge) $\Longrightarrow$ feu1 := orange []
(feu1 = rouge $\wedge$ feu2 = vert) $\Longrightarrow$ feu2 := orange []
(feu1 = rouge $\wedge$ feu2 = orange) $\Longrightarrow$ feu1, feu2 := vert, rouge []
(feu1 = orange $\wedge$ feu2 = rouge) $\Longrightarrow$ feu1, feu2 := rouge, vert)
$\Leftrightarrow$
prd((feu1 = vert $\wedge$ feu2 = rouge) $\Longrightarrow$ feu1 := orange) $\vee$
prd((feu1 = rouge $\wedge$ feu2 = vert) $\Longrightarrow$ feu2 := orange) $\vee$
prd((feu1 = rouge $\wedge$ feu2 = orange) $\Longrightarrow$ feu1, feu2 := vert, rouge) $\vee$
prd((feu1 = orange $\wedge$ feu2 = rouge) $\Longrightarrow$ feu1, feu2 := rouge, vert)
$\Leftrightarrow$
((feu1 = vert $\wedge$ feu2 = rouge) $\wedge$ prd(feu1 := orange)) $\vee$
((feu1 = rouge $\wedge$ feu2 = vert) $\wedge$ prd(feu2 := orange)) $\vee$
((feu1 = rouge $\wedge$ feu2 = orange) $\wedge$ prd(feu1, feu2 := vert, rouge)) $\vee$
((feu1 = orange $\wedge$ feu2 = rouge) $\wedge$ prd(feu1, feu2 := rouge, vert))
$\Leftrightarrow$
((feu1 = vert $\wedge$ feu2 = rouge) $\wedge$ prd(feu1 := orange)) $\vee$
((feu1 = rouge $\wedge$ feu2 = vert) $\wedge$ prd(feu2 := orange)) $\vee$
((feu1 = rouge $\wedge$ feu2 = orange) $\wedge$ prd(feu1 := vert $\|$ feu2 := rouge)) $\vee$
((feu1 = orange $\wedge$ feu2 = rouge) $\wedge$ prd(feu1 := rouge $\|$ feu2 := vert))
$\Leftrightarrow$
((feu1 = vert $\wedge$ feu2 = rouge) $\wedge$ feu1' = orange) $\vee$
((feu1 = rouge $\wedge$ feu2 = vert) $\wedge$ feu2' = orange) $\vee$
((feu1 = rouge $\wedge$ feu2 = orange) $\wedge$ (feu1' = vert $\wedge$ feu2' = rouge)) $\vee$
((feu1 = orange $\wedge$ feu2 = rouge) $\wedge$ (feu1' = rouge $\wedge$ feu2' = vert))

La contrainte OCL se déduit de cette post-condition :

**Context** *CARREFOUR : :ChangerFeux()* **post :**
*((feu1@pre = COUL_Value : :vert* **and** *feu2@pre = COUL_Value : :rouge)* **and**
*feu1 = COUL_Value : :orange)* **or**
*((feu1@pre = COUL_Value : :rouge* **and** *feu2@pre = COUL_Value : :vert)* **and**
*feu2 = COUL_Value : :orange)* **or**
*((feu1@pre = COUL_Value : :rouge* **and** *feu2@pre = COUL_Value : :orange)* **and**
*(feu1 = COUL_Value : :vert* **and** *feu2 = COUL_Value : :rouge))* **or**
*((feu1@pre = COUL_Value : :orange* **and** *feu2@pre = COUL_Value : :rouge)* **and**
*(feu1 = COUL_Value : :rouge* **and** *feu2 = COUL_Value : :vert))*

FIG. 16.13 – Extraction de la post-condition de l'opération *ChangerFeux*

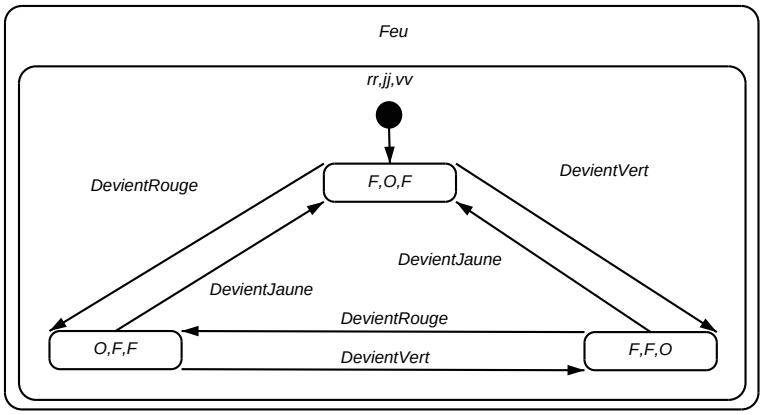


FIG. 16.14 – Diagramme d'états-transitions extrait de la spécification *Feu*

Dans ce diagramme d'états-transitions (fig 16.14), on peut valider le comporterment de la spécification du *Feu* relativement au cahier des charges. On peut ainsi valider les règles 1 et surtout 2 énoncées dans la section 16.2.2.

Pour les mêmes raisons que dans la figure 16.14 nous avons remplacé les valeurs *rouge, vert* et *orange* respectivement par *R, V* et *J*.

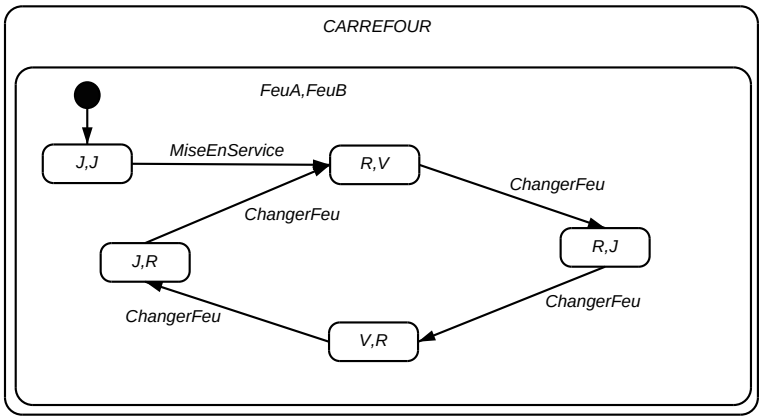


FIG. 16.15 – Diagramme d'états-transitions extrait de la spécification *CARREFOUR*

Sur ce diagramme d'états-transitions (fig 16.15) on peut valider la règle 3 de la section 16.2.2.

### 16.4.4 Diagramme de séquence

Le diagramme de séquence de la figure 16.16 correspond à l'extraction depuis la spécification *CARREFOUR_imp*. Il permet de visualiser les appels des opérations symbolisés par des messages entre les lignes de vie des objets.

### 16.4.5 Diagramme d'activités

Le diagramme d'activités de la figure 16.17 est extrait de la spécification *MAIN_imp*. L'instruction *WHILE* de la spécification est représentée sous la forme d'une boucle d'états (de

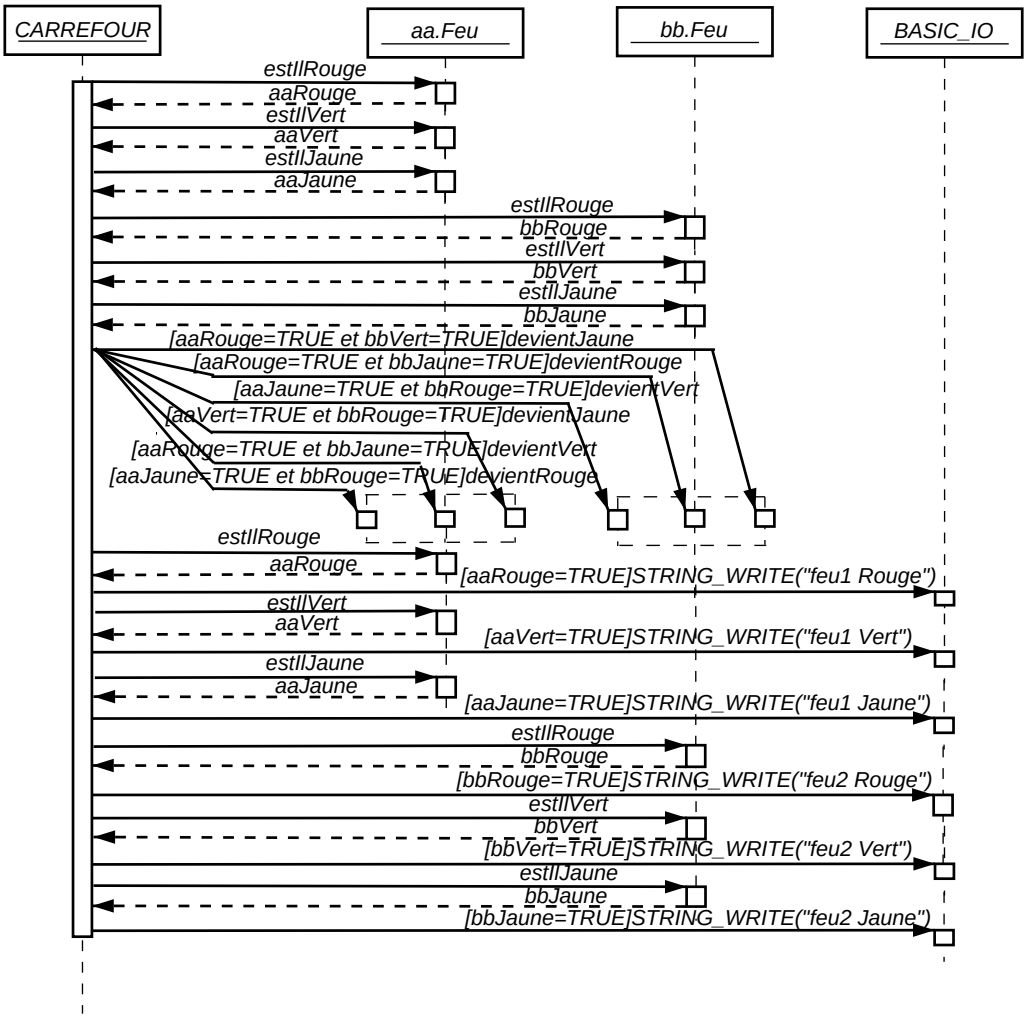


FIG. 16.16 – Diagramme de séquence extrait de la spécification *CARREFOUR_imp*

*E7* à *E11*). Chaque état représente la fin d'exécution de chaque instruction.

## 16.5 Traduction en Java

Dans cette traduction (fig 16.18,16.19 et 16.20) le type de base utilisé est le type **`short`** car les valeurs des variables utilisées (*rr, vv, oo* et *ii*) des implémentations sont comprises dans l'intervalle [-32768, 32767]. Dans le cas ou la variable *ii* est utilisée différemment (nombre d'itérations élevé de la boucle) il sera nécessaire d'être prudent quant à l'utilisation du type de base (l'outil de preuve devra être paramétré par ce type pour les constantes *MININT* et *MAXINT*).

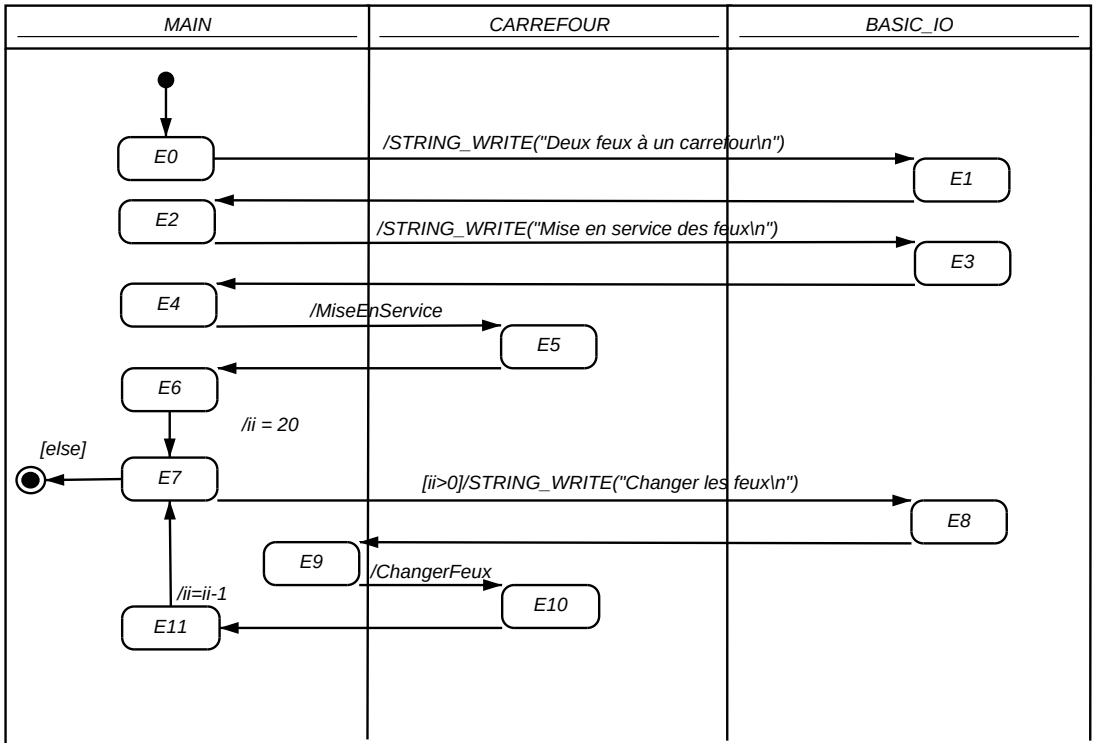


FIG. 16.17 – Diagramme d'activités extrait de la spécification *MAIN_imp*

```
import java.util.Arrays;

class MAIN {

  static void INITIALISATION() {
    BASIC_IO.INITIALISATION();
    CARREFOUR.INITIALISATION();
  }

  public static void main (String Args[]) {
    INITIALISATION();
    short ii;
    BASIC_IO.STRING_WRITE("Deux feux a un carrefour\n");
    BASIC_IO.STRING_WRITE("Mise en service des feux\n");
    CARREFOUR.MiseEnService();
    ii = (short)((short)20);
    while (ii > (short)0) {
      BASIC_IO.STRING_WRITE("Changer les feux\n");
      CARREFOUR.ChangerFeux();
      ii = (short)(ii - (short)1);
    }
  }
}
```

FIG. 16.18 – Traduction du composant principal en Java

```
import java.util.Arrays;

class CARREFOUR {
  static Feu aa; static Feu bb; static boolean hh;

  static void INITIALISATION() {
    bb = new Feu(); aa = new Feu();
    aa.devientOrange(); bb.devientOrange();
  }

  public static void MiseEnService () {
    aa.devientRouge(); bb.devientVert();
  }

  public static void ChangerFeux () {
    short aaRouge; short aaVert; short aaOrange;
    short bbRouge; short bbVert; short bbOrange;
    aaRouge = aa.estIlRouge(); aaVert = aa.estIlVert();
    aaOrange = aa.estIlOrange(); bbRouge = bb.estIlRouge();
    bbVert = bb.estIlVert(); bbOrange = bb.estIlOrange();
    if (aaVert == 1 && bbRouge == 1) { aa.devientOrange(); }
    else if (aaRouge == 1 && bbVert == 1) { bb.devientOrange(); }
    else if (aaRouge == 1 && bbOrange == 1) {
      aa.devientVert(); bb.devientRouge();
    }
    else if (aaOrange == 1 && bbRouge == 1) {
      aa.devientRouge(); bb.devientVert();
    }
    aaRouge = aa.estIlRouge();
    if (aaRouge == 1) { BASIC_IO.STRING_WRITE("feu 1 Rouge"); }
    aaVert = aa.estIlVert();
    if (aaVert == 1) { BASIC_IO.STRING_WRITE("feu 1 Vert"); }
    aaOrange = aa.estIlOrange();
    if (aaOrange == 1) { BASIC_IO.STRING_WRITE("feu 1 Orange"); }
    bbRouge = bb.estIlRouge();
    if (bbRouge == 1) { BASIC_IO.STRING_WRITE(", feu 2 Rouge\n"); }
    bbVert = bb.estIlVert();
    if (bbVert == 1) { BASIC_IO.STRING_WRITE(", feu 2 Vert\n"); }
    bbOrange = bb.estIlOrange();
    if (bbOrange == 1) { BASIC_IO.STRING_WRITE(", feu 2 Orange\n"); }
  }
}
```

FIG. 16.19 – Traduction du composant *CARREFOUR* en Java

```
import java.util.Arrays;

class Feu {

  short rr; short oo; short vv;

  public Feu () {INITIALISATION();}
  void INITIALISATION() {
    rr = (short)(ETAT.off); oo = (short)(ETAT.on); vv = (short)(ETAT.off);
  }
  public short estIlRouge () {
    short estRouge = 0;
    if ((rr == ETAT.on)) {estRouge = 1;} else estRouge = 0;
    return estRouge;
  }
  public short estIlOrange () {
    short estOrange = 0;
    if ((oo == ETAT.on)) {estOrange = 1;} else {estOrange = 0;}
    return estOrange;
  }
  public short estIlVert () {
    short estVert = 0;
    if ((vv == ETAT.on)) {estVert = 1;} else {estVert = 0;}
    return estVert;
  }
  public void devientRouge () {
    rr = (short)(ETAT.on); oo = (short)(ETAT.off); vv = (short)(ETAT.off);
  }
  public void devientOrange () {
    rr = (short)(ETAT.off); oo = (short)(ETAT.on); vv = (short)(ETAT.off);
  }
  public void devientVert () {
    rr = (short)(ETAT.off); oo = (short)(ETAT.off); vv = (short)(ETAT.on);
  }
  public void changerCouleur () {
    short cc;
    cc = (short)(rr); rr = (short)(oo); oo = (short)(vv); vv = (short)(cc);
  }
}
```

FIG. 16.20 – Traduction du composant *Feu* en Java

# Cinquième partie

# Conclusion

# Chapitre 17

# Conclusion et perspectives

Tout d'abord rappelons brièvement que nos objectifs initiaux visaient à la mise en place de principes d'intégration des méthodes formelles et semi-formelles et à la traduction de spécifications formelles vers les langages à objets. Tout ceci devait être implanté dans le cycle de développement des logiciels. L'intégration des méthodes formelles et semi-formelles peut être réalisée sous trois formes :

- intégration par extension,
- intégration par dérivation,
- intégration par extraction.

Nous avons choisi l'intégration par extraction pour les raisons suivantes :

- L'intégration par extension ne permet pas la conservation des avantages et points forts des deux méthodes comme la preuve ou le raffinement.
- L'intégration par dérivation d'une part est délicate du fait que le langage (notation) source est plus riche que la méthode cible (formelle). D'autre part les spécifications obtenues ne sont pas complètes, il est donc nécessaire d'y ajouter certains éléments puis de les prouver.
- L'intégration par extraction, que nous avons choisi, prend en compte ce phénomène car dans cette stratégie on commence par écrire la spécification en langage B en laissant libre cours aux concepteurs. Le principe est d'en extraire par la suite les vues UML permettant de mieux la documenter et la comprendre.

Le premier but de cette stratégie, dans le cycle de développement des logiciels, est la validation des spécifications par rapport au cahier des charges. Les méthodes formelles paraissent souvent d'accès délicat. Le premier obstacle auquel on se heurte est souvent une difficulté de lecture. L'extraction de vues UML permet, via les vues UML, une meilleure lisibilité ce qui permet une meilleure compréhension pour les non initiés et donc une utilisation par un public plus large.

Le deuxième objectif de notre contribution est le gain de temps et donc de productivité lié à l'utilisation des méthodes formelles. En effet le temps de développement imparti à l'utilisation des méthodes formelles est largement supérieur à celui de l'utilisation des méthodes semi-formelles. Il est donc nécessaire pour une meilleure utilisation de réduire les temps qui lui sont impartis dans le développement complet jusqu'à l'implantation. Les temps de développement des phases d'analyse et d'écriture des spécifications étant incompressibles il faudrait en gagner sur le reste du cycle de développement. Nous remplissons cet objectif pour les raisons suivantes :

- l'aide à la validation des spécifications par rapport au cahier des charges par extraction de vues UML,
- la réduction des phases de tests à cause de la preuve des spécifications formelles,
- la traduction automatique des spécifications formelles vers les langages à objets tels que Java et C#.

## 17.1 Contributions

Notre contribution est divisée en deux parties. La première est basée sur l'utilisation des notations à objets pour valider et documenter la spécification formelle. Le modèle proposé préconise l'utilisation conjointe de la méthode B et des notations à objets. Elle est constituée d'un ensemble de schémas d'extraction de vues UML depuis les spécifications B. Les diagrammes traités dans cette stratégie sont au nombre de quatre :

- diagramme de classes,
- diagramme d'états-transitions,
- diagramme de séquence,
- diagramme d'activités.

De plus nous traduisons les propriétés complexes de l'invariant et les pré et post-conditions des opérations en langage OCL ce qui permet de les adapter aux diagrammes de classes donc d'uniformiser le résultat complet de l'extraction. Ces méthodes d'extraction sont réalisées à partir d'un sous-ensemble du langage B dont les arbres et records sont exclus. D'autre part l'extraction de diagrammes d'états-transitions est élaborée à partir du même sous-ensemble restreint aux spécifications dont au moins une variable est typée à l'aide d'ensembles énumérés. La deuxième contribution est une traduction à partir d'implémentations B écrites à l'aide du sous-ensemble B0 (langage d'implémentation de B) vers les langages compilables à objets tels que Java et C#. Ces deux contributions étant réalisées autour de B, nous avons choisi de factoriser les phases préliminaires d'analyse syntaxique et de vérification de types des projets B dans une plate-forme commune.

### 17.1.1 Extraction de vues UML

Le cycle de développement du langage B (fig 2.1) est complet. Il permet de gérer entièrement le cycle de développement logiciel (fig 1.1) à partir de la phase Conception/Spécification. Notre contribution à l'utilisation conjointe des méthodes formelles et semi-formelles par extraction de vues UML et de contraintes OCL doit donc intervenir sur les trois étapes suivantes du cycle de développement d'applications (fig 1.1) :

- spécification / conception,
- implémentation / codage,
- implantation / test.

Les différentes extractions énoncées précédemment remplissent parfaitement leur rôle vis à vis des étapes de spécification/conception et d'implémentation/codage. En effet la phase de spécification/conception, réalisée à l'aide des machines abstraites qui définissent les concepts généraux du système, est validée relativement au cahier des charges par les extractions relatives aux abstractions comme :

- les diagrammes de classes,
- les contraintes OCL,
- les diagrammes d'états-transitions.

La phase d'implémentation/codage, quant à elle, est validée toujours relativement au cahier des charges par les extractions d'autres diagrammes comme :

- les diagrammes de séquence,
- les diagrammes d'activités.

Ceci permet de valider le comportement détaillé du système car ces diagrammes sont extraits depuis les implémentations des spécifications B. Néanmoins l'apport de validation n'a pas autant d'importance que pour la partie abstraite et conceptuelle des spécifications car le langage B0 est très proche des langages impératifs traditionnels. Par conséquent il est compris par un plus grand nombre.

### 17.1.2 Traduction vers les langages à objets Java et C#

La traduction de spécifications B en langage à objets de type Java que nous avons mise en place peut s'appliquer dans deux contextes différents. Le premier est appliqué aux logiciels pour les systèmes informatisés standards. Le deuxième est relatif aux applications embarquées plus particulièrement pour JavaCard. Dans les deux cas, nous avons restreint la traduction en bloquant l'instanciation par le type STRING de machines prenant comme paramètres des ensembles. Nous avons maintenu la notion de modularité B en la substituant par la notion de classe. Nous avons optimisé les classes Java générées en terme de mémoire en utilisant des variables et méthodes de type statique et en permettant de choisir le type d'entier de base pour la plate-forme cible. En l'état actuel cette possibilité devra être utilisée avec précaution car il faut paramétrer les constantes MININT et MAXINT avant de prouver les spécifications B. La traduction Java standard permet l'utilisation du langage B0 dans sa totalité hormis la restriction citée précédemment tandis que la traduction spécifique pour JavaCard est plus contraignante à cause de ses spécificités en termes de mémoire et de structure. L'extension pour du code C# est elle aussi terminée. Il serait possible de choisir d'autres langages à objets disposant de la généricité sur les types primitifs comme C++ afin de générer les différents types d'entiers.

### 17.1.3 La plate-forme jBTools

Nous avons présenté la plate-forme jBTools. Cette plate-forme est composée de fonctionnalités communes d'analyse syntaxique, sémantique et de vérification de typage. Le résultat est enregistré dans des fichiers XML (un par composant B). Il est ensuite exploitable par la BOL (B Object Library) sur laquelle viennent se greffer les outils de la plate-forme comme :

- un traducteur vers les langages à objets,
- un extracteur de diagrammes UML.

Actuellement nous disposons d'un extracteur de diagrammes UML partiellement achevé et d'un traducteur finalisé. D'autre part notre plate-forme peut s'interfacer avec l'outil ProB[1]. ProB [67] est un outil de model-checking qui utilise les fichiers XML générés par notre plate-forme. Cet outil complémentaire à ceux développés au sein du LIFC permettrait un achèvement total de la plate-forme par rapport au cycle de développement B (Fig 2.1).

[1] http ://www.ecs.soton.ac.uk/ mal/systems/prob.html

### 17.1.4 Synthèse des contributions

Les intégrations de méthodes formelles et semi-formelles antérieures réalisées le plus souvent par dérivation mettaient en jeu deux diagrammes : les diagrammes de classes et d'états-transitions. Notre extraction de vues UML depuis une spécification B répond aux exigences du domaine car elle comprend quatre diagrammes dont ceux de classes et d'états-transitions en plus de l'extraction des propriétés complexes de l'invariant en contraintes OCL. Notre extraction fait apparaître aussi le traitement des parties concrètes des modèles comme les diagrammes de séquence et d'activités extraits à partir des implémentations. La traduction des spécifications B vers les langages à objets est ciblée sur un large domaine puisqu'elle est adaptée autant aux logiciels standards qu'à l'embarqué comme les JavaCards. Elle optimise autant que possible le code généré mais doit être guidée par l'utilisateur dans certains cas. Nous avons créé un outil regroupant d'une part l'extraction de vues UML (contraintes OCL comprises) et d'autre part la traduction vers Java et C#. Ceci a permis de factoriser les développements d'analyse syntaxique et sémantique des projets B. Cet outil a été étendu par l'interfaçage avec l'outil ProB cité précédemment. Cette plate-forme répond à des besoins industriels de par les méthodes conceptuelles utilisées pour une meilleure contribution de chacune d'entre elles :

- la preuve des méthodes formelles diminue les phases de tests,
- les méthodes semi-formelles apportent une meilleure lisibilité du projet,

et par les cibles des différents traducteurs développés :

- Java standard : application logiciel standard,
- Java pour JavaCard : applications embarquées.

## 17.2 Perspectives

Les perspectives de ces travaux de thèse sont de quatre ordres. La première est d'obtenir une convergence entre l'intégration par dérivation et l'intégration par extraction. La deuxième vise à améliorer les principes d'extraction des diagrammes d'états-transitions. La troisième est de terminer la plate-forme JBTOOLS. La quatrième vise à fusionner les principes de traductions et d'extractions.

### 17.2.1 Démarche itérative entre l'extraction et la dérivation

Actuellement les deux démarches d'intégration des méthodes formelles et semi-formelles divergent, l'utilisation successive de l'une et de l'autre, quelque soit l'ordre d'utilisation dérivation et extraction ou vice-versa, ne permet pas de revenir au modèle de départ. Les causes de cette divergence sont à mettre en relation avec le but de ces deux démarches. En effet le modèle formel résultat de la dérivation doit être prouvé après son obtention tandis que le modèle semi-formel extrait, explicite et documente la spécification formelle. D'autre part les deux méthodes, semi-formelle et formelle, ont des fondements très différents. Néanmoins il serait intéressant d'étudier la convergence de ces deux techniques d'intégration dans l'optique d'une démarche itérative c'est-à-dire d'obtenir le même modèle après l'application des deux transformations successivement sur le modèle de départ. Ainsi les spécialistes de la méthode B et ceux de la notation UML pourront tour à tour enrichir le modèle après chaque itération comme dans les travaux de Okalas Ossami, Souquières et Jacquot [30] sachant que chaque modèle (B ou UML) sera ensuite transcrit dans l'autre notation par extraction ou dérivation.

### 17.2.2 Amélioration de l'extraction des diagrammes d'états-transitions

L'extraction des diagrammes d'états-transitions souffre de quelques problèmes dus aux restrictions sur le sous-ensemble du langage B considéré. En effet nous avons limité l'extraction aux variables typées avec des ensembles énumérés. Dans l'optique de créer un outil, il sera certainement possible d'améliorer la qualité du résultat de l'extraction.

#### 17.2.2.1 Production de diagrammes d'états-transitions pour des ensembles quelconques

Il est peut être possible d'élargir le sous-ensemble du langage B utilisé dans cette extraction. Les relations utilisées, fonctions, bijections, surjections et injections totales ou partielles sont souvent définies en B par un ensemble d'entiers vers un autre ensemble d'entiers. Le problème rencontré lors de l'extraction de diagrammes d'états-transitions depuis ce genre de spécifications est l'explosion combinatoire qui en découle car en réalité ces ensembles reviennent à des gros ensembles énumérés traités dans notre méthode. Il serait possible de réduire l'explosion combinatoire en paramétrant une restriction sur une seule valeur du domaine de la relation. Néanmoins il est nécessaire de connaître la sémantique de la spécification donc seul l'utilisateur de l'application générant l'extraction peut effectuer ce genre de manipulation. D'autre part il sera nécessaire d'utiliser au maximum les caractéristiques de modularité de la méthode B car lorsqu'il y a trop de variables, le diagramme sera quand même illisible et inexploitable.

#### 17.2.2.2 Amélioration de la vue extraite par sélection de variables

Actuellement nous extrayons un diagramme d'états-transitions par machine abstraite du projet B. La validation de la spécification par rapport aux règles de gestion du cahier des charges est réalisée à l'aide de ce diagramme. Les règles sont traduites dans l'invariant par des propriétés que le système doit respecter. Elles sont exprimées en fonction des variables. Durant l'extraction il faudrait pouvoir choisir les variables à montrer dans le diagramme de manière à les isoler du reste du diagramme complet de la spécification qui est obtenu dans notre démarche. L'avantage de les isoler permet de mettre en relief certains aspects du comportement du système et de le comparer aux règles définies dans l'invariant. L'outil qui permet d'extraire le diagramme d'états-transitions pourra sélectionner les variables utilisées en n-uplet dans les règles de gestion du cahier des charges. On obtient un diagramme relatif aux variables mises en jeu par les règles de gestion par calcul du produit synchronisé des diagrammes de chaque variable. Ainsi les états seront des n-uplet de valeurs des variables contenues dans le prédicat considéré. En effet chaque règle de gestion du cahier des charges est traduite par un prédicat (ce n'est pas un prédicat de typage) de l'invariant. Il faut prendre chaque prédicat complexe de l'invariant, extraire les identifiants de variables compris à l'intérieur et exprimer un diagramme d'états-transitions en fonction uniquement de ces variables. Dans le cas contraire, c'est-à-dire s'il n'existe pas de propriétés complexes de l'invariant mettant en jeu plusieurs variables, les diagrammes montrant le comportement individuel de chaque variable seront suffisants. Ceci simplifierait la tâche de validation des spécifieurs UML.

### 17.2.3 Achèvement de la plate-forme JBTOOLS

Les traducteurs sont terminés au contraire des différentes extractions. A l'heure actuelle la plate-forme JBTOOLS permet d'extraire les diagrammes de classes et certaines contraintes OCL. Dans un avenir proche, il serait intéressant de pouvoir extraire non seulement l'ensemble des contraintes OCL mais aussi les différents diagrammes montrant le comportement du système comme :

- diagramme d'états-transitions,
- diagramme de séquence,
- diagramme d'activités.

Il serait au moins nécessaire de permettre l'extraction des diagrammes d'états-transitions et d'activités puisque l'appel d'opérations des diagrammes d'activités recouvre presque totalement les communications des diagrammes de séquence.

### 17.2.4 Fusion des deux méthodes Extraction/Traduction

Une éventuelle fusion entre les méthodes d'extraction de diagrammes de classes et de traduction vers les langages à objets pourrait être entreprise étant donné que les deux langages cibles sont très proches car ils possèdent tous deux la notion d'objet. En effet il existe peu de différence entre les diagrammes de classes et les données décrites dans les classes Java. La différence entre les deux langages réside dans la notation. En effet la notation UML ne permet pas de décrire de la même manière que la notation Java ou C# les extractions de relations. Les relations sont traduites en Java et C# par des tableaux dont le domaine de définition de l'indice (pour les tableaux à une dimension) est défini par le domaine de la relation et le domaine de définition des valeurs par le codomaine de la relation. Or à travers l'extraction les relations deviennent des agrégations. Il sera peut-être possible à l'aide des nouvelles versions d'UML, dont la version 2 bientôt disponible fait partie, d'uniformiser ces deux aspects. Néanmoins la nature des informations traitées par la traduction vers les langages à objets et l'extraction de diagrammes est différente car la première est purement concrète à l'opposé de la deuxième qui est très abstraite. Nous ne sommes pas au même niveau entre l'extraction de vues UML réalisée à partir du sous-ensemble du langage B abstrait et la traduction vers Java et C# engendrée à partir du sous-ensemble d'implémentation du langage B, le B0.

### 17.2.5 Validation de spécifications relativement au cahier des charges

Dans ce document nous avons pour l'instant défini la validation des spécifications B par extraction de vues UML et vérification de la conformité des diagrammes par rapport au cahier des charges de manière informelle par un spécialiste de la notation UML. Nous avons aussi décrit le fait qu'actuellement plusieurs modèles UML pouvaient spécifier le même système et que ni l'un ni l'autre solution n'était forcément le meilleur. Les différences de notations sont fonction des habitudes de travail des auteurs de la spécification. Il serait intéressant de réaliser une étude pratique dans le domaine industriel concernant la validation de modèles UML par rapport au cahier des charges.

# Bibliographie


[1] J-R. Abrial. Extending B without changing it (for developping distributed systems). In Henri Habrias, editor, *1st Conference on the B method, Nantes(F)*, pages 169–190, November 1996. ISBN 2-906082-25-2.

[2] J.-R. Abrial. Constructions d'automatismes industriels avec B. In ONERA-CERT, editor, *Congrès AFADL'97*, 1997.

[3] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[4] J.R Abrial and L. Mussat. Introducing dynamic constraints in b. In *Conference on the B method*, volume LNCS 1393, pages 83–128, France, April 1998. Springer Verlag.

[5] P. Behm. Dévelopement formel des logiciels sécuritaires de METEOR. In *1rst conférence on the B method*, pages 3–10, 1996.

[6] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. METEOR : A succesful application of B in a large project. In *World Congres on Formal Methods*, pages 369–387, 1999.

[7] B. Bermond. Des spécifications B vers le modèle UML des données, septembre 2002. Rapport de stage, DESS Développement de Logiciels Sûrs, ALSTOM - CNAM, Paris(F).

[8] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embeded C Programs. In *FME'2003*, volume LNCS 2805, pages 94–113, Pisa Italie, 2003. Springer-Verlag.

[9] M. Bidoit and P. Mosses. *CASL User Manual - Introduction to Using the Common Algebraic Specification Language*, volume 2900. Springer-Verlag, 2004. ISBN : 3-540-20766-X.

[10] G. Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc, second edition, 1994. ISBN 0-8053-5340-2.

[11] P.P. Chen. The entity-relationship model : Toward a unified view of data. In *ACM Transaction on Database Systems*, pages 9–36, mars 1976.

[12] Z. Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley, 2000. ISBN 0-201-70329-7.

[13] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object-oriented Development : The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ, Object-oriented series edition, 1994.

[14] T. Coquant and G. Huet. *The calculus of construction*. Inf. Comp., 1988. 76(2).

[15] B Core. *B-Toolkit User's Manual, (UK) Ltd*. Oxford (UK), 1996. Release 3.2.

[16] Bert D. and Cave F. Construction of Finite Labelled Transition Systems from B Abstract Systems. In *IFM2000*, volume LNCS 1945, pages 235–254, Dagstuhl, 2000. Springer-Verlag.

[17] C. DaSilva, B. Dehbonei, and F. Mejia. Formal specification in the development of industrial applications : The subway speed control mechanism. In *FORTE'91*, pages 207–221, 1991.

[18] J. Dick and J. Loubersac. Integrating Structured and Formal's Methods : A Visual Approach to VDM. In *3nd International Conference ESEC'91*, pages 37–59. Springer-Verlag, October 1991.

[19] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood, Cliffs, 1976. ISBN : 013215871X.

[20] R. Duke, P. Kin, G. Rose, and G. Smith. *The ObjectZ Specification Language.* Departement of computer science. University of Queensland, Austria, Software Verification Research Center, avril 1991.

[21] R. Duke, G. Rose, and G. Smith. ObjectZ : a Specification Language Advocated for the Description of Standards. Technical Report 94-45, University of Queensland, Software Verification Research Center, Queensland, Australia, December 1994.

[22] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Integrating OMT and ObjectZ. In *BCS FACS/EROS ROOM Workshop*, London UK, 1997.

[23] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Translating the OMT dynamic model into ObjectZ. In *11th Conference of Z users*, pages 347–366, Berlin Germany, 1998.

[24] E.H. Durr and J. Van Katwijk. VDM++ a Formal Specification Language for Object-oriented Design. In *Computer Systems and software Engineering*, pages 214–219. IEEE Society Press, 1992.

[25] F. Bouquet et B. Legeard et N. Vacelet. Un format fédérateur pour l'évaluation de spécifications formelles en programmation logique avec contraintes. In Mireille Ducassé, editor, *Programmation en Logique avec Contraintes, JFPLC'2003*, volume Hors serie RSTI (Revue des Sciences et Technologie de l'Information), pages 203–216, Amiens, juin 2003. Hermes Lavoisier.

[26] M. C. Gaudel et B. Marre et F. Schlienger et G. Bernot. *Précis de génie logiciel.* Masson, 1996. ISBN 2-225-85189-1.

[27] A. Hammad et B. Tatibouët et J.C. Voisinet. De la spécification b vers les diagrammes d'états-transitions UML. In *7ième MCSEAI'2002 (Maghrebian Conference of Sofware Engineering and Intelligence Artificial)*, pages 133–143, mai 2002.

[28] F. Badeau et D. Bert et S. Boulmé et M.-L. Potet et N. Stouls et L. Voisin. Traduction de B vers des langages de programmation : points de vue du projet BOM. In *AFADL'2003*, pages 87–102, Rennes, 2003. IRISA.

[29] J.F. Monin et G. Huet. *Comprendre les méthodes formelles : Panorama et outils logiques.* Masson, 1996. ISBN 2-225-85304-5.

[30] D. D. Okalas Ossami et J. Souquières et J.-P. Jacquot. Opérations de construction de spécifications multi-vues uml et b. In *AFADL'04*, pages 115–129, juin 2004.

[31] B. Tatibouët et J.C. Voisinet. Plugin B, type-checking, génération de code java et visualisation de projets B en HTML dans la nouvelle plate-forme jbtools. In *Journées du groupe B du GDR ALP*, CNAM - Paris, France, 13-14 juin 2002.

[32] H. Fekih et L. Jemni et S. Merz. Transformation des spécifications B en des diagrammes UML. In *AFADL'2004, 7th Conférence : Approches formelles pour l'aide au développement de logiciels*, pages 131–146, Besançon, France, juin 16-18 2004.

[33] N. Stouls et M.-L. Potet. Explicitation du contrôle de développements B événementiel. In *AFADL'04*, pages 13–28, juin 2004.

[34] N. Stouls et M.-L. Potet et X. Morselli. GénéSyst : Génération d'un système de transitions étiquetées à partir d'une spécification B événementiel. In *AFADL'04*, pages 317–320, juin 2004.

[35] P.-A. Muller et N. Gertner. *Modélisation objet avec UML*. Eyrolles, 2000. ISBN 2-212-09122-2.

[36] R. Marcano et N. Lévy. Transformation d'annotations OCL en expressions B. In *AFADL'2001*, pages 39–49, 2001.

[37] P. Behm et P. Desforges et F. Mejia. Application de la méthode B dans l'industrie ferroviaire. In *Application des Techniques Formelles au Logiciel*, pages 59–88. Observatoire français des techniques avancées, Lavoisier TEC et DOC, 1997. Chap 3.

[38] C. Attiogbé et P. Poizat et G. Salaün. Intégration de données formelles dans les diagrammes d'états d'UML. In J.-M. Jézéquel, editor, *AFADL 2003*, pages 3–17, Rennes, 2003.

[39] F. Ambert et S. Chemin et B. Legeard. Intégration de domaines à variables dans un solveur de contraintes ensemblistes. In Mireille Ducassé, editor, *Programmation en Logique avec Contraintes, JFPLC'2003*, volume Hors serie RSTI (Revue des Sciences et Technologie de l'Information), pages 217–222, Amiens, juin 2003. Hermes Lavoisier.

[40] P. Facon, R. Laleau, and H.P. Nguyen. Mapping Object Diagrams into B Specifications. In Springer-Verlag, editor, *Methods Integration Workshop, Electronic Workshops in Computing (eWiC), Leeds (UK)*, March 1996.

[41] C. Fischer. *Combination and implementation of processes and Data from CSP-OZ to Java*. PhD thesis, Univerité d'Oldenburg, Allemagne, January 2000.

[42] R. France, J.M. Bruel, M. Larrondo-Petrie, and E. Grant. Rigorous object-oriented modeling : Integrating formal and informal notations. In *6th International AMAST Conference*, pages 216–230. LNCS 1349, ISBN 3-540-63888-1, 1997.

[43] A.J.M. Van Gasteren. *On the shape of mathematical arguments*. Springer-Verlag, LNCS 445, 1990. ISBN : 3540528490.

[44] J.A. Goguen, J. Meseguer, and D. Plaisted. *Programming with Parametrized Abstract Objects in OBJ*. Theory and Practice of Software Technology, North-Holland, 1983.

[45] D. Gries. *The Science of Programming*. Springer-Verlag, New-York USA, 1981.

[46] J.V. Guttag, J.J. Horning, and J.M. Wing. *The Larch familly of specification languages*. IEEE Software 2(5), 1985.

[47] H. Habrias. *Introduction à la spécification*. Masson, 1993. ISBN :2-225-82768-0.

[48] H. Habrias. *Spécification formelle avec B*. Hermes science Publications, 2001. ISBN 2-7462-0302-2.

[49] A. Hammad, B. Tatibouët, J.C Voisinet, and W. Weiping. From a B Specification to UML Statechart Diagrams. In *4th International Conference on Formal Engineering Methods, ICFEM'2002*, Shangai, China, October 21-25 2002. LNCS.

[50] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*, volume 12, pages 576–580, 1969.

[52] IFAD[2]. The VDM++ to Java Code Generator, 2000.

[53] IFAD[2]. The Dynamic Link Facility for VDM++, 2001.

[54] IFAD[2]. The VDM C++ Library, 2001.

[55] IFAD[2]. The VDM-SL to C++ Code Generator, 2001.

[56] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering : A Use case Driven Appraoch.* Addison-Wesley, 1992. ISBN 0201544350.

[57] W. Johnston and G. Rose. Guidelines for the manual conversion of ObjectZ to C++, 1993. Technical Report 9314, University of Queensland, Software Verification Research Center, Queensland, Australia, 1993.

[58] C. B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[59] K. Lano. Z++, an Object-oriented Extension to Z. In *Z user workshop, Workshop in computing*, pages 151–172, Oxford (UK), 1991. J. E. Nicholls, Springer-Verlag.

[60] K. Lano. *The B Langage and Method : a Guide to Practical Formal Development.* Springer-Verlag, 1996. ISBN 3-540-76033-4.

[61] K. Lano and H. Haughthon. Improving the Process of System Specification and Refinment in B. In *6th Refinment workshop International AMAST Conference.* D. Till Editor, Springer-Verlag, 1994.

[62] K. Lano, H. Houghton, and P. Wheeler. *Formal Methods and Object Technology : Integrating Formal and Structured Methods in Object-Oriented System Development.* Springer-Verlag, 1996. ISBN 3-540-19977-2.

[63] H. Ledang. Des cas d'utilisation à une spécification B. In *AFADL'01*, pages 131–140, Nancy, 2001.

[64] H. Ledang. *Traduction systématique de spécifications UML en B.* PhD thesis, LORIA - Université Nancy 2, Nancy - France, novembre 2002.

[65] B. Legeard, F. Peureux, and M. Utting. A comparison of the BTT and TTF/Z test-generation methods. In *The $2^{nd}$ International Z and B Conference (ZB'02)*, volume LNCS 2272, pages 309–329, Grenoble, France, January 2002. Springer-Verlag.

[66] X. Leroy. A modular module system, April 1996. Research report 2866, INRIA.

[67] M. Leuschel and M. Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[68] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurent Systems. In *Fomal Methods in System Design*, volume 6, pages 1–36, 1995.

[69] A. Malioukov. An Object-based Approach to the B Formal Method. In *B'98 Recent Advances in the Development and Use of the B Method*, pages 162–181, Montpellier France, 1998. LNCS 1393.

[2]http ://www.ifad.dk

[70] A. Mammar. Utilisation de graphes pour la spécification de formules relationnelles, septembre 1998. Rapport de stage de DEA IIE-INT-EVE Conception des systèmes informatiques avancés.

[71] G. Mariano. *Evaluation de logiciels critiques développés par la méthode B : Une approche quantitative.* PhD thesis, Université de Valenciennes et du Hainaut-Cambrésis, décembre 1997.

[72] B. Meyer. *Conception et programmation orientées objet.* Eyrolles, 2000. ISBN : 2212091117.

[73] E. Meyer. *Développement formels par objets : utilisation conjointe de B et d'UML.* PhD thesis, LORIA - Université Nancy 2, Nancy - France, mars 2001.

[74] E. Meyer and J. Souquières. A Systematic Approach to Transform OMT Diagrams to a B Specification. In *FM'99*, pages 875–895. LNCS 1708 vol. 1, 1999.

[75] Sun Microsystems. Java card$^{TM}$ 2.2 specification and development kit - url : http ://java.sun.com/products/javacard/.

[76] N. Nagui-Raïss. A formal software specification tool using the entity-relationship model. In Pericles Loucopoulos, editor, *Entity-Relationship Approach - ER'94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, U.K., Proceedings*, LNCS 881, pages 315–332. Springer, 1994.

[77] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles.* PhD thesis, Conservatoire national des arts et métiers, Paris - France, décembre 1998.

[78] S. Owre, J. M. Rushby, and N. Shankar. PVS : A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[79] D. Petit. *Génération automatique de composant logiciels sûrs à partir de spécifications formelles B.* PhD thesis, Université de Valenciennes et du Hainaut-Cambrésis, décembre 2003.

[80] M.-L. Potet. Etude du passage de paramètres dans la méthode B - optimisation mémoire, 2001. Rapport technique du projet BOM. URL : http ://lifc.univ-fcomte.fr/~tatibouet/WEBBOM.

[81] A. Requet and G. Bossu. Embedding Formally Proved Code in a Smart Card : Converting B to C. In *Formal Engineering Methods 2000*, pages 15–22, 2000.

[82] J. Rumbaugh, M. Blaha, W. Premerlani, F. EDDY, and W. Lorensen. *Object-oriented modelling and design.* Prentice-Hall, New Jersey, 1991.

[83] P. Schnoebelen. *Vérification de logiciels.* Vuibert, 1999. ISBN : 2-7117-8646-3.

[84] E. Sekerinski. Graphical design of reactive systems. In *B'98 : Recent advances in the development and use of the B-Method*, pages 182–197, LNCS 1393, Montpellier - France, April 1998.

[85] R. Sethi. *Programming languages : Concepts and constructs.* Addison Wesley, 1989.

[86] M. Shroff and R.B. France. Towards a Formalization of UML Class Structures in Z. In *COMPSAC'97*, 1997.

[87] J. Sifakis. A unified approach for studying the properties of transition systems. In *Theoritical Computer Science*, volume 18, pages 227–258, 1982.

[88] Rational Software Corporation. *Unified Modeling Language.* http ://www.rational.com/uml, Santa Clara USA, 1997.

[89] I. Sommerville. *Le génie logiciel.* Addison-Wesley, 1992. ISBN 2-87908-033-9.

[90] J.M. Spivey. *The Z notation : A reference Manual.* 2nd edition, Prentice-Hall, Englewood Cliffs, NJ., 1992.

[91] Steria. AtelierB : Preuves et exemples, 2001. Atelier B version 3.6.

[92] Stéria. *Atelier B, Manuel d'utilisateur- Technologies de l'information.* Aix-en-Provence (F), 1998. Version 3.5.

[93] Stéria. Manuel de référence du language B. Technical report, Stéria, 2000. version 1.8.4.

[94] B. Tatibouet. Evaluation du traducteur Java. Rapport Technique LIFC, Besançon, août 2002.

[95] B. Tatibouet. La plate-forme jBTools. In *Journées outils pour et autour de B*, INRETS-Lille, France, 13-14 juin 2001.

[96] B. Tatibouet, A. Hammad, and J.C. Voisinet. From an Abstract B Specification to UML Class Diagrams. In *2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002)*, pages 5–10, December 2002. ISBN 0-9727186-0-5.

[97] B. Tatibouet, A. Requet, J.C. Voisinet, and A. Hammad. Java Card Code Generation from B Specifications. In *5th International Conference on Formal Engineering Methods (ICFEM'2003)*, pages 306–318, Singapore, November 2003. LNCS 2885.

[98] B. Tatibouet and J.C. Voisinet. JBTOOLS and B2UML : a Platform and a Tool to Provide an UML Class Diagram from a B Specification. In *ICSSEA'2001, 14th International Conference on Software and Systems Engineering and Their Applications*, volume Vol 2, Paris, France, December 4-6 2001. Formal Methods Session.

[99] B. Tatibouet and J.C. Voisinet. Generating Statecharts from B Specifications. In *16th International Conference on Software and Systems Engineering and their applications (ICSSEA'2003)*, volume 1, December 2003.

[100] J.C. Voisinet. Traducteur B vers Java : Les grandes lignes de l'implémentation en vue d'une extension à d'autres langages. Rapport Technique LIFC, Besançon, août 2002.

[101] J.C. Voisinet. Traducteur B vers Java : Manuel utilisateur. Rapport Technique LIFC, Besançon, août 2002.

[102] J.C. Voisinet, B. Tatibouet, and A. Hammad. jBTools : An Experimental Platform for the Formal B Method. In *Principles and Practice of Programming in Java (PPPJ'02)*, pages 137–140, Trinity College, Dublin, Ireland, June 2002.

[103] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling with UML.* Addison-Wesley, 1999. ISBN 0-201-37940-6.

## Résumé

Les travaux de recherche dans le domaine du génie logiciel ont permis de concevoir deux types d'approche, les méthodes formelles qui permettent d'écrire et de vérifier une spécification d'un système à l'aide d'une notation issue des mathématiques et les méthodes semi-formelles orientées objet qui permettent de modéliser un système en utilisant des vues graphiques. Le point de départ de nos travaux repose sur une méthode formelle : la méthode B. Nous apportons une contribution dans deux étapes du développement logiciel. Le premier est l'intégration des deux types de méthode par extraction de vues graphiques UML depuis une spécification formelle B. Cette extraction est arrivée à maturité puisqu'elle met en jeu les diagrammes de classes, d'états-transitions et les contraintes OCL comme la plupart des intégrations déjà réalisées telles que la dérivation de notations graphiques vers les méthodes formelles. D'autre part nous avons aussi défini l'extraction de diagrammes plus concrets comme ceux de séquence et d'activités. La deuxième contribution de nos travaux est la traduction des spécifications B en langage compilable et exécutable. Cette traduction est ciblée sur un large domaine car elle est appliquée aux logiciels standard à l'aide de Java et C# mais aussi à l'embarqué en utilisant Java pour JavaCard. Une plate-forme regroupant les outils relatifs à ces deux contributions a été développée afin de factoriser les développements communs d'analyse syntaxique et sémantique des spécifications B.



**Mots clefs :** Génie logiciel, Extraction, UML, Méthode B, Traduction, Validation, Java et C#.



## Summary

The research works about software engineering allows to think up two approach types, the formal methods which allows to write and verify a system specification with a notation from mathematics and the semi-formal methods object-oriented which allows to model a system using graphic views. The departure of our works is the B formal method. We provide a contribution in two states of the software development. The first is the integration of the two method types by extraction of UML graphics views from a B formal specification. This extraction happen to maturity because it deals with state-charts, classes diagrams and OCL constraints like most integration works already done as graphic notation derivation to formal methods. In addition we also defined the extraction of concrete diagrams like sequence and activity ones. The second contribution of our works is the translation of B specifications to compilable and executable languages. This translation is targeted on a large domain because it is applied on standard software with Java and C# but also on embedded application with Java for JavaCard. A plate-forme merging the two contributions relatives tools was developed in order to factorize the common developments of syntactic and semantic analysis.



**Key words :** Software engineering, Extraction, UML, B Method, Translation, Validation, Java and C#.